UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

TRABAJO FIN DE MÁSTER

Desarrollo de sondas de monitorización virtuales over-switch para entornos SDN

Máster Universitario en Ingeniería de Telecomunicación

Autor: Pegado Boureghida, Mª Teresa
Tutor: Ramos de Santiago, Javier
Ponente: López de Vergara Méndez, Jorge E

FECHA: Septiembre, 2017

Desarrollo de sondas de monitorización virtuales over-switch para entornos SDN

AUTOR: Pegado Boureghida, Mª Teresa
TUTOR: Ramos de Santiago, Javier
PONENTE: López de Vergara Méndez, Jorge E

HPCN
Dpto. de Tecnología Electrónica y de las Telecomunicaciones
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Septiembre de 2017

RESUMEN

En los últimos años ha surgido con fuerza el concepto de la virtualización de funciones de red (NFV, Network Function Virtualization). La idea principal es prescindir del uso de middleboxes hardware e implementar las funciones de red virtualmente en software, de modo que se puedan instanciar dinámicamente en función de la demanda (VFNaaS). El mantenimiento y actualización se hace más simple y barato y permite la adaptación dinámica al tráfico de red.

Otro concepto emergente son las redes definidas por software (SDN, Software Defined Networks), mediante la separación de los planos de control y datos permite abstraerse del hardware subyacente y aporta programabilidad de la red y una visión centralizada de la misma. SDN se complementan perfectamente con el paradigma de las funciones virtuales de red, simplificando la orquestación y el encadenamiento de las distintas funciones de red.

En este trabajo, se han estudiado Mininet y Docker, como tecnologías de virtualización, junto con Open vSwitch para la integración de funciones virtuales de red para monitorización. El objetivo es instanciar fácil y dinámicamente las funciones de monitoreo. Se han desarrollado dos funciones software para la monitorización HTTP y para la detección de ataques por inundación de SYN. Se han llevado a cabo pruebas de rendimiento y funcionalidad. El troughput dentro de un contenedor se ha medido enviando a una tasa de 10Gbps, el troughput entre dos contenedores también se ha medido. Los test de funcionalidad han probado la movilidad de las NFV y verificado los datos obtenidos.

PALABRAS CLAVE

NFV, SDN, Mininet, Docker, Open vSwitch, monitorización, redes de comunicaciones.

ABSTRACT

In recent years, the concept of Network Function Virtualization (NFV) has grown strongly. The main idea is to dispense the use of hardware middleboxes and to implement the network functions virtually in software, so that it can be deployed dynamically and on demand (VNFaaS). Management and update process becomes simpler and cheaper, and it allows for the dynamic adaptation to network traffic.

Another emerging concept is Software Defined Networks (SDN), that, by means of the separation of the control and data planes it allows abstracting from the underlying hardware. Additionally, allows for network programmability and provides a centralized view of it. SDN complements perfectly the NFV paradigm, simplifying the orquestation and the chaining of the network functions.

In this project, Mininet and Docker have been studied as virtualization technologies along with Open vSwitch for the integration of software network functions for monitoring. The aim is to instantiate easily and dynamically the monitoring functions. Two software functions have been developed one for HTTP monitoring and the other for the detection of SYN-flood attacks. Performance and functionality tests have been carried out. The troughput inside de container have been measured forwarding traffic at a rate of 10Gbps, the troughput between two containers have also been measured. Functionality test have probed the NFV mobility and checked the reported data.

KEYWORDS

NFV, SDN, Mininet, Docker, Open vSwitch, monitoring, communication networks.

AGRADECIMIENTOS

Quiero agradecer a Javier Ramos por tutorizar este trabajo de fin de máster, así como mi trabajo de fin de grado.

También agradecer a mis padres por su apoyo incondicional.

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1 INTRODUCCIÓN

1.1 MOTIVACIÓN

En el marco de la cuarta revolución industrial [1], donde todo estará conectado por el denominado Internet de las Cosas (IoT por sus siglas en inglés) que conectará a Internet elementos variados de los que se obtendrá información y acceso remoto, se generarán ingentes cantidades de datos que se analizarán con técnicas de Big Data. La infraestructura necesaria para proporcionar estas funcionalidades se sustenta sobre servicios y aplicaciones basadas en computación en la nube, la cual proporciona Software como Servicio (SaaS por sus siglas en inglés), Plataforma como Servicio (PaaS por sus siglas en inglés) e Infraestructura como Servicio (IaaS por sus siglas en inglés). Este desarrollo ha sido posible gracias principalmente a las técnicas de **virtualización**. Asimismo, a día de hoy la movilidad se ha convertido en un aspecto clave con tendencias como BYOD (Bring Your Own Device) en la que los usuarios emplean cada vez más dispositivos móviles personales, como teléfonos inteligentes, tabletas y portátiles para el acceso a las redes corporativas y empresariales.

El panorama anteriormente descrito requiere que la infraestructura de los centros de datos y las empresas, sean escalables, flexibles y confiables. Por lo que la rigidez de la arquitectura de red tradicional es un inconveniente a la hora de desplegar nuevos servicios y características. Dicha rigidez se debe principalmente a la dependencia de conjunto de software y hardware físico heterogéneo, especializado y, generalmente, propietario. En los últimos años, se han propuesto tecnologías como **SDN** (Software Defined Network) y **NFV** (Network Function Virtualization), que permiten una abstracción del hardware subyacente permitiendo alcanzar los requerimientos de escalabilidad, flexibilidad y confiabilidad necesarios en la infraestructura IT actual reduciendo el coste y la dependencia de soluciones propietarias.

Las redes actuales están compuestas por un conjunto heterogéneo de dispositivos. Los elementos más comunes son los routers y switches, que se encargan de encaminar y conmutar el tráfico de red de manera correcta desde un origen a un destino. Sin embargo, también nos encontramos con otros elementos denominados middleboxes, como pueden ser firewalls o cortafuegos, Sistemas de Detección de Intrusión (IDS) o Traductores de Direcciones de Red (NAT) entre otros. Cada middlebox suele implementar una o varias funciones de red como pueden ser la aplicación de políticas de seguridad, el almacenamiento remoto de datos o la monitorización de la red.

En el ámbito de la gestión y la operación de redes, la monitorización es una actividad clave. La monitorización permite analizar los patrones de tráfico y caracterizar la red, obtener medidas para asegurar el rendimiento de la red a través de la caracterización de parámetros de QoS (Quality of Service) o poder aplicar políticas de seguridad consistentes, para la detección y mitigación de ataques.

Los middleboxes suelen ser elementos hardware, habitualmente de carácter propietario, lo cual conlleva un coste alto y añade cierta dificultad a la hora de inter-operar con elementos de distintos fabricantes. Las funciones que implementan los middleboxes son esenciales a día de hoy, pero su correcta incorporación en una red resulta complicada. Dicha complicación reside en planificar cuidadosamente la topología para incorporar estos

middleboxes e instalar reglas manualmente en los dispositivos de red para enviar el tráfico a través de la secuencia deseada de middleboxes, en lo que se denomina comúnmente encadenamiento de servicios, como se explica en [2].

El despliegue de nuevos servicios como la virtualización de infraestructuras en entornos Cloud, como vemos en [3], conlleva una reducción del CAPEX y OPEX, que son respectivamente los gastos iniciales y de mantenimiento que suponen a una empresa la implantación de un nuevo recurso o tecnología. Al compartir recursos de computación, redes y almacenamiento, se reducen dichos gastos. En este entorno en los últimos años ha surgido con fuerza el concepto de la virtualización de funciones de red (NFV, Network Function Virtualization). La idea principal es implementar las funciones de red (anteriormente implementadas por middleboxes) en software que se pueda ejecutar en máquinas virtuales (o elementos similares) dinámicamente en función de la demanda (VFNaaS), eliminando de esta manera, la necesidad de instalar y mantener hardware especializado. Además, la virtualización de funciones de red permite un mantenimiento y actualización más simple y barato que permite que los servicios o funciones de red evolucionen a la vez que el tráfico que circula por las redes.

Por otro lado, la nueva tendencia hacía las redes definidas por software o SDN (Software Defined Network) simplifica la gestión e implantación de servicios e infraestructura en el panorama descrito anteriormente. De hecho, ya no existen diferencias entre los que hemos denominados elementos de red básicos como son los switches, routers, o hubs. En este nuevo escenario, tenemos un único elemento de red, que se puede denominar “white box” y la funcionalidad se programa en un elemento denominado “controlador” [4]. La comunicación entre el controlador y el “white box” se realiza mediante diferentes protocolos o mecanismos, siendo el estándar OpenFlow [5] el más usado a día de hoy.

Aunque ciertas funciones de red pueden programarse en el controlador, las redes definidas por software se complementan perfectamente con el paradigma de las funciones virtuales de red (NFV), simplificando la orquestación y el encadenamiento de las distintas funciones de red virtualizadas.

En la actualidad las compañías están empezando a migrar hacia estos nuevos paradigmas en la arquitectura de red, que tienden más hacia el desarrollo software y la virtualización. Para que esta migración sea paulatina, en el mercado se empiezan a encontrar switches híbridos, como puede ser el caso de los switches de Intel [6]. Éstos permiten una migración controlada entre la arquitectura tradicional de red y una arquitectura SDN basada en OpenFlow, ya que soportan la gestión de tráfico usando dicho protocolo y la gestión de tráfico basada en protocolos tradicionales. Además, estos equipos permiten la programación del controlador SDN directamente sobre la arquitectura del equipo (over-switch programming) sin requerir ningún equipo extra. Esta característica nos permite incluir las funciones virtuales de red en el propio equipo de red, teniendo los beneficios ya descritos de reducción de costes, al no necesitar incorporar a la red un nuevo dispositivo, aunque sea “off-the-self”. Además, también se simplifica la orquestación del tráfico a través de las funciones de red al poderlo instalar en el propio equipo de red y se eliminan problemas de retardos adicionales al no tener que desviar el tráfico hacia equipos externos en los cuales se encuentra la función de red como ocurre con los middleboxes tradicionales.

Como ejemplo de esta realidad, está el caso de Microsoft Azure, en un post del blog de Microsoft Azure realizado por el CTO Mark Russinovich sobre el Open Networking Summit realizado en 2015 [1], se explica que, debido al crecimiento exponencial de Azure, doblando su almacenamiento y computo cada seis meses, tuvieron que implantar SDN y NFV en sus centros de datos.

1.2 OBJETIVOS

Una vez expuesto el contexto, en este trabajo de final de máster se plantea como objetivo **el desarrollo flexible de funciones virtuales de red para monitorización integradas en el propio equipo de red para entornos SDN**. Estas funciones constituirán unas sondas software que realizarán monitorización HTTP y de ataques de denegación basados en SYN flooding como ejemplo de su aplicabilidad. Además, las sondas desarrolladas deberán tener la capacidad de moverse transparentemente entre los equipos de comunicaciones y otros equipos utilizados como sondas de red tradicionales. De forma más concreta los objetivos serán:

- Analizar diferentes tecnologías de virtualización como Docker y Mininet.
- Desarrollar sondas software empleando dichas tecnologías de virtualización.
- Permitir el movimiento de las sondas de monitorización.
- Evaluar el rendimiento de las herramientas y sistemas desarrollados.

La finalidad es el desarrollo de herramientas que se adapten a las nuevas tendencias de migración hacia el software y la virtualización que se ven en la actualidad y que pretenden reducir costes y simplificar el contexto de la arquitectura de red, respectivamente.

Este trabajo se ha servido de las competencias desarrolladas en las Materia 1.3: Planificación, Gestión y Aplicaciones de Redes de Comunicaciones del Nivel 1: Tecnologías de Telecomunicación, de acuerdo a la terminología empleada en la Memoria de Verificación del Máster de Ingeniería de Telecomunicación de la Universidad Autónoma de Madrid [7]. Las competencias enunciadas en dicho apartado son:

- Capacidad para diseñar redes de comunicaciones utilizando protocolos y arquitecturas estándares, aplicar criterios de eficiencia y escalabilidad al diseño de convergencia de redes y servicios.
- Capacidad de enunciar, comprender y aplicar los principios de diseño de la arquitectura de dispositivos de comunicaciones.
- Capacidad para enunciar los principios de arquitectura de gestión de redes y aplicarlos al diseño de sistemas de gestión de redes basados en arquitectura estándar.
- Capacidad para enunciar los principios que rigen el funcionamiento y organización de Internet.
- Capacidad de enunciar y aplicar las tecnologías y protocolos de Internet de nueva generación.
- Capacidad de diseñar servicios basados en las tecnologías y protocolo de Internet.

[1] https://azure.microsoft.com/es-es/blog/report-from-open-networking-summit-achieving-hyper-scale-with-software-defined-networking/

1.3 ORGANIZACIÓN DE LA MEMORIA

Este documento se organiza en cinco capítulos. El primer capítulo sirve de Introducción, mostrando la motivación y objetivos de este trabajo de fin de master, junto con la organización del documento. En el segundo capítulo, Estado del Arte, se muestra un estudio de las técnicas de virtualización, del protocolo OpenFlow, virtualización de funciones de red (NFV), redes definidas por software (SDN), además de un análisis de tecnologías similares encontradas en la literatura. En el tercer capítulo, Diseño y Desarrollo, se muestra cómo se han desarrollado las sondas de monitorización y su integración con las tecnologías de virtualización. En el cuarto capítulo, Pruebas y Resultados, se presentan las pruebas y resultados relevantes obtenidos para validar el funcionamiento de los elementos desarrollados, así como su rendimiento. Por último, se presenta el capítulo 5 con las Conclusiones más relevantes de este trabajo, así como las mejoras que podrían añadirse en un futuro.

2 ESTADO DEL ARTE

En este capítulo se van a explicar diferentes conceptos, que son necesarios para comprender las diferentes tecnologías utilizadas en este trabajo.

2.1 TÉCNICAS DE VIRTUALIZACIÓN

La virtualización es un elemento clave que permite exprimir la capacidad de computo de los servidores además de proporcionar aislamiento y seguridad entre aplicaciones. La virtualización hace posible entre otras cosas la computación en la nube, aunque también se usa en los simuladores y emuladores de red, además de ser una herramienta muy útil para los desarrolladores de software a la hora de probar sus aplicaciones.

Las técnicas de virtualización se pueden dividir típicamente en las basadas en Hipervisores y las basadas en contenedores.

La virtualización basada en hipervisores o Monitor de Máquina Virtual (VMM por sus siglas en inglés) [8], permiten a la máquina física particionarse en varias máquinas virtuales, cada una ejecutando su propio sistema operativo sobre un hardware virtual propio. El hipervisor trabaja a bajo nivel, ofreciendo una capa de abstracción sobre el sistema anfitrión. Un ejemplo de este caso es la ejecución de máquinas virtuales basadas en Windows sobre Linux.

Como se ve en la Figura 2-1, existen dos tipos de hipervisores. Los hipervisores de tipo 1 o nativos operan sobre el hardware anfitrión, mientras que los hipervisores de tipo 2 o anfitriones, operan sobre el sistema operativo anfitrión.

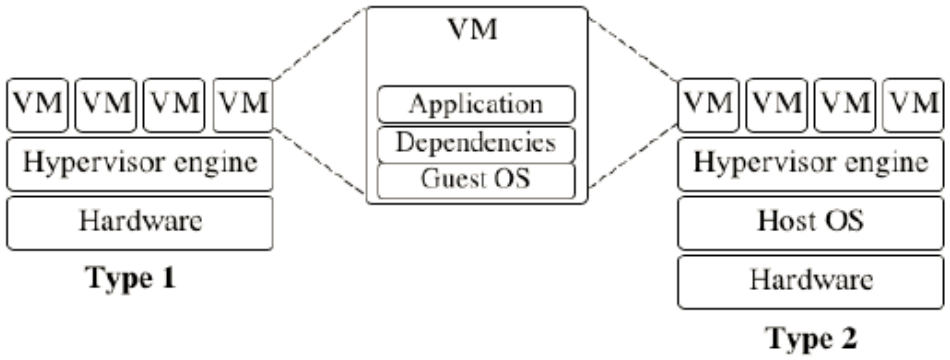

Figura 2-1: Virtualización basada en Hipervisores [8].

El segundo tipo de virtualización disponible a día de hoy es la virtualización basada en contenedores la cual ha surgido con fuerza en los últimos años. En este tipo de virtualización los contenedores se ejecutan sobre el núcleo del sistema operativo anfitrión y uno o más procesos se ejecutan dentro de cada contenedor. De este modo se aíslan los procesos de cada contendor y se proporciona la ilusión de tener sistemas independientes, como se ve en la Figura 2-2.

La virtualización basada en contenedores hace uso de los llamados *namespaces* [9]. Los *namespaces* son una funcionalidad incluida a partir de la versión del *kernel* de Linux 2.2.6. Los *namespaces* encapsulan un recurso del sistema global en una abstracción que hace que parezca que los procesos asociados al *namespace* tienen su propia instancia aislada de los recursos globales. Cambios en los recursos globales son visibles para otros procesos que son miembros del *namespace*, pero son invisibles para el resto de procesos. En Linux están implementados *namespaces* para el sistema de ficheros, IPC, red, usuarios y PID entre otros.

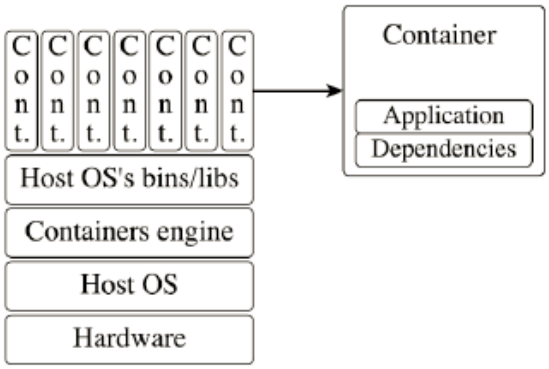

Figura 2-2: Virtualización en contenedores [8]

La virtualización basada en contenedores es una alternativa ligera a la virtualización basada en hipervisores, ya que la virtualización del hardware y la ejecución del sistema operativo completo en el caso de los hipervisores sobrecarga mucho el equipo anfitrión y en muchas ocasiones no es necesarios virtualizar un sistema operativo completo sino solo una parte del mismo (por ejemplo, la pila de red).

2.2 VIRTUALIZACIÓN DE FUNCIONES DE RED (NFV)

De acuerdo a la definición proporcionada por la ETSI (European Telecommunications Standards Institute), las funciones virtuales de red o NFV *"evolucionan la tecnología estándar de virtualización para concentrar varios tipos de equipamiento de red en servidores industriales de alto volumen, switches y almacenamiento, que pueden estar localizados en centros de datos, nodos de red y en las instalaciones del usuario final. Las funciones virtuales de red se implementan en software que puede ejecutarse en múltiples servidores y que pueden moverse o instanciarse en varias localizaciones de la red según se necesite, sin la necesidad de instalar nuevos equipos*" [10].

Añadir un nuevo servicio, en las redes de hoy en día se está convirtiendo en algo realmente complicado debido a la naturaleza propietaria de las aplicaciones de red existentes, el coste de ofrecer el espacio y energía para una variedad de middleboxes y la falta de profesionales cualificados para integrar y mantener estos servicios. Las funciones virtuales prometen aliviar estos problemas, junto a otras tecnologías como las redes definidas por software (SDN) y la computación en la nube [11].

Con las funciones virtuales de red, al eliminar el elemento físico, la estructura de la red cambia, prescindiendo de dispositivos hardware especializados, y dependiendo únicamente de almacenamiento estándar, switches y servidores de propósito general sobre los que se ejecutan aplicaciones virtuales, como se ve en la Figura 2-3.

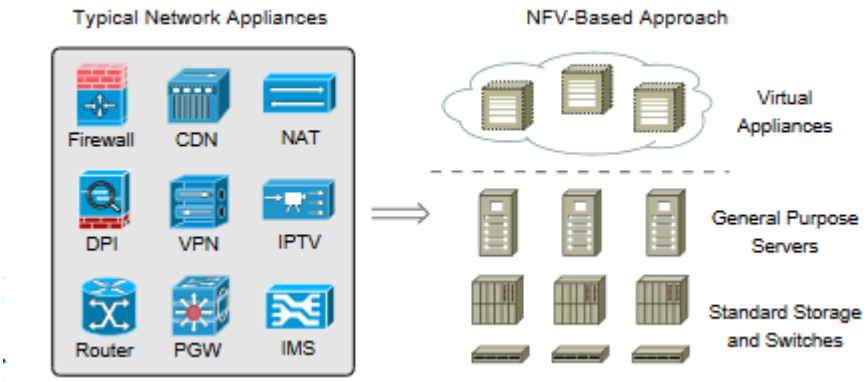

Figura 2-3: Redes tradicionales vs aproximación basada en NFV [11]

Las funciones virtuales (VNFs en la Figura 2-4) se ejecutan sobre la Infraestructura NFV (NFVI), que consta de:

- Recursos hardware, como servidores, que tiene los recursos físicos como CPU, almacenamiento y de red.
- Capa de virtualización, como puede ser un hipervisor, o monitor de máquina virtual. Este es el software que gestiona los recursos físicos, proporcionando el entorno virtual sobre el que se ejecutan las máquinas virtuales huésped.
- Máquina virtual huésped, es un software que emula la arquitectura y funcionalidades de una plataforma física en la que se ejecuta la función de red (VNFs).

Luego el Orquestador y Gestor de NFV, automatiza el despliegue, operación gestión y coordinación de las VNFs y los recursos de la NFVI.

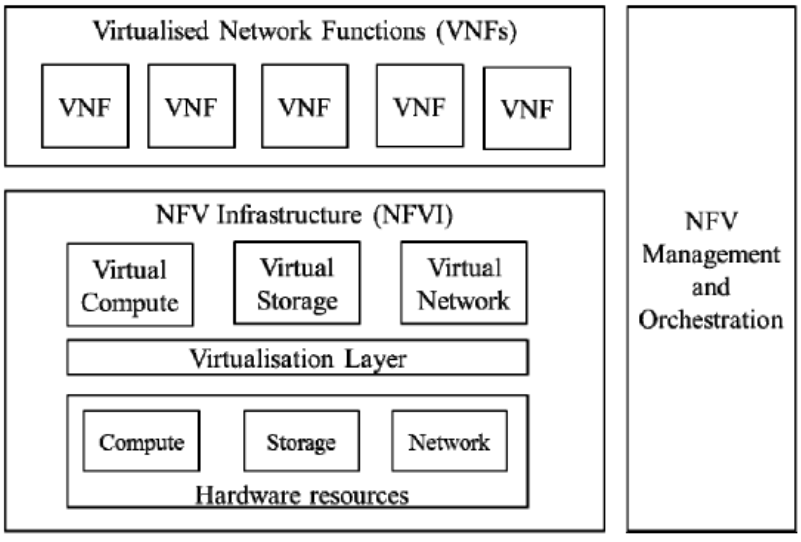

Figura 2-4:Arquitectura NFV [12].

Para desplegar una arquitectura de telecomunicación con nodos que permitan la integración de funciones virtuales de red, con un rendimiento similar a las redes tradicionales con dispositivos hardware especializado, hay que tener una serie de factores en cuenta, como se explica en [13]. Por un lado, las funciones de red se implementan en servidores de propósito general (*commodity* servers), que han sido diseñados para computación, pero no específicamente para procesamiento de paquetes de red. La virtualización del servidor puede degradar el rendimiento, ya que al emular el sistema operativo huésped hace que se puede generar cuellos de botella en los procesos de entrada/salida (I/O) a pesar del uso de tecnologías de virtualización asistida por hardware como Intel VT-d o AMD-V.

El manejo de paquetes de red por parte del kernel del sistema operativo se ha basado hasta hace poco en un modelo basado en interrupciones, sin embargo, Linux NAPI (new API) introduce el concepto de polling sobre el modelo basado en interrupciones para mitigar el número de interrupciones. Las interrupciones de los dispositivos asociados a las interfaces de red degradan el throughput obtenido cuando la tasa de paquetes por segundo es alta (en entornos de 10 Gb/s y superiores). Otro aspecto relacionado con el hardware que hace el procesamiento de paquetes ineficiente es la cantidad de copias que se realizan entre áreas de memoria. El kernel de Linux copia los paquetes recibidos a varias áreas de memoria para pasar los paquetes a las aplicaciones, esto causa degradación en el rendimiento de I/O de red [14].

En este escenario se han desarrollado múltiples motores de captura que han tratado de solucionar estos problemas. En concreto, Intel ha desarrollado Intel Data Plane Development Kit (DPDK), que proporciona un entorno de manejo de paquetes red que permite la copia directa de los paquetes de red al espacio de usuario de las aplicaciones, sin interrupciones hardware, usando los recursos de CPU exclusivamente para el manejo de paquetes. Haciendo uso de estos motores de captura, la programación de funciones virtuales (VNFs) se simplifica y puede realizarse desde equipos de propósito general e incluso switches que puedan ejecutar un sistema operativo GNU Linux.

A la hora de añadir una función de red mediante un middlebox, normalmente se añaden estos middleboxes en el camino directo entre dos puntos finales, teniendo que realizar cambios en la infraestructura de red. Al virtualizar las funciones y mover la implementación software a centros de datos, hay que redirigir el tráfico hacía el centro de datos donde se encuentre la función de red causando retardos añadidos y sobrecarga en las tablas de rutas [11]. Alternativamente se pude dar el caso en el que, puesto que la función virtual de red se puede instanciar dinámicamente en diferentes puntos de la red (siempre que el hardware lo permita), la función virtual de red se pueda colocar en un punto de interés donde la redirección del tráfico no suponga retardos añadidos y sobrecargas en las tablas de rutas. En este sentido la creación de funciones virtuales dentro de equipamientos de red como switches es la estrategia más óptima.

2.3 Redes Definidas por Software (SDN)

De acuerdo a la Open Network Fundation (ONF) [15], un consorcio industrial sin ánimo de lucro que está liderando el avance de SDN y estandarizando elementos críticos de la arquitectura SDN, las Redes Definidas por Software, son un nuevo paradigma dentro de las redes de comunicaciones, que se basa en la separación entre los planos de control y datos en los equipos de red y es directamente programable. Esta migración del control, desde los dispositivos individuales de red, a dispositivos computacionalmente accesibles, permite que

la infraestructura subyacente sea abstraída para que aplicaciones y servicios de red puedan tratar a la red como una entidad lógica o virtual.

En contraposición a la definición que se ha dado de SDN, tenemos las redes tradicionales, en las cuales, como se explica en [16], los planos de control y datos están combinados en los nodos de red. El plano de control es responsable de la configuración del nodo y de programar los caminos y reglas que se usarán para los flujos de datos. Luego el envío de los datos a nivel de hardware se basa en esa información de control. En este enfoque tradicional, una vez se ha definido una política sobre el tráfico, la única manera de hacer un ajuste es cambiando la configuración de los dispositivos. Esto supone una limitación para los operadores de redes, a la hora de escalar la red, como respuesta a las demandas de tráfico cambiantes y al aumento del uso de dispositivos móviles.

Además, otra de las principales limitaciones, que ha llevado al planteamiento de esta nueva arquitectura de red, es la complejidad de las redes actuales. Los protocolos actuales de red proporcionan un mayor rendimiento y fiabilidad, conectividad más amplia, y seguridad más estricta. El problema es que cada protocolo da solución a un problema específico, sin el beneficio de ningún tipo de abstracción.

La Figura 2-5 representa una vista lógica de la arquitectura SDN, consta de tres capas la capa de infraestructura o datos, la capa de control y la capa de aplicación. Entre la capa de datos y la de control hay una interfaz que se denomina Southbound-API. Esta interfaz permite la comunicación entre ambas capas, y generalmente se implementa usando el protocolo estándar OpenFlow. Entre la capa de control y la de aplicación, está la denominada Northbound-API, permite un nivel más de abstracción permitiendo una programación de alto nivel para la realización de aplicaciones complejas.

El controlador de una SDN es una entidad lógicamente centralizada, esto quiere decir, que puede consistir en múltiples instancias físicas o virtuales, pero se comporta como un único componente. Una red tiene un controlador que mantiene un estado global de una red o fragmento de red particular. De este modo, las empresas y operadores ganan control sobre toda la red desde un único punto lógico, lo que simplifica enormemente el diseño, despliegue, operación y mantenimiento de la red. SDN también simplifica los dispositivos de red en sí, ya que no necesitan entender y procesar miles de protocolos sino simplemente aceptar las instrucciones del controlador SDN lo cual favorece la adaptabilidad de las redes al tráfico que transportan.

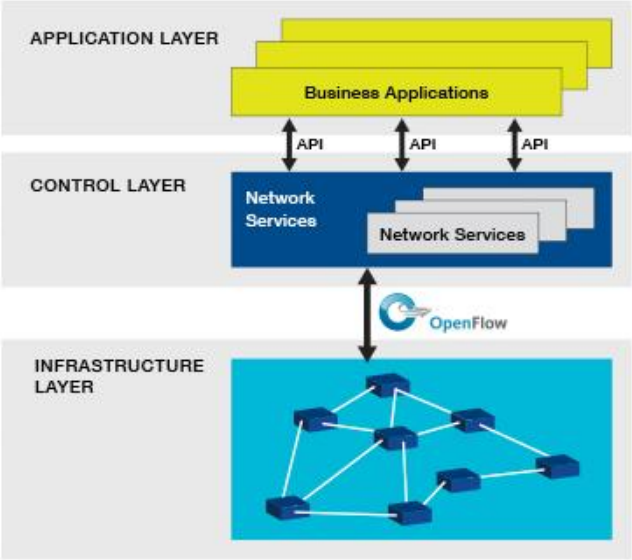

Figura 2-5: Arquitectura SDN[2].

Un switch SDN, se trata de un elemento de red genérico sin ninguna inteligencia, que simplemente reenvía, añade o modifica algún campo de los paquetes de acuerdo a una tabla de flujos que modifica el controlador. Un flujo agrupa paquetes en función de una serie de campos comunes como pueden ser direcciones IP puertos etc. En la Tabla 2-1 se especifican los campos concretos que se emplean en el protocolo OpenFlow para definir un flujo. El switch puede tener disponible una o más tablas de flujos y una tabla de grupo. Haciendo uso de dichas tablas, se lleva a cabo la revisión y reenvío de paquetes. La comunicación con los controladores se realiza generalmente usando uno o más canales OpenFlow que usan mecanismos de cifrado para proveer seguridad.

Cada vez que llega un paquete a un switch SDN, se comprueba si existe una entrada en alguna de sus tablas de flujos. En caso de encontrar una coincidencia, las instrucciones asociadas con la entrada en cuestión son ejecutadas. Si no se encuentran coincidencias en una tabla de flujos, se envía el paquete al controlador el cual decidirá qué hacer con el paquete y modificará consecuentemente la tabla de flujos del switch [17].

OpenFlow es uno de los protocolos más extendidos y el único estándar de comunicación entre las capas de control y de datos en una arquitectura SDN. OpenFlow se comenzó a desarrollar por un grupo de ingenieros de la Universidad de Standford entre 2008 y 2009. La Open Networking Foundation (ONF) ahora gestiona dicho estándar.

OpenFlow permite el acceso directo y la manipulación del plano de datos en dispositivos de red compatibles con el protocolo OpenFlow tanto físicos como virtuales. Emplea el concepto de flujo para identificar el tráfico basándose en reglas predefinidas de coincidencia, que pueden ser estáticas o programadas dinámicamente por el software de control de SDN. Esto permite un control granular, permitiendo que la red responda en tiempo real a cambios.

Los datos para realizar la coincidencia del paquete en OpenFlow, puede ser cualquier combinación de los siguientes campos:

[2] Fuente: https://www.opennetworking.org/sdn-resources/sdn-definition

Campos del switch	Ethernet	IPv4	TCP/UDP
Puerto de entrada	Dirección origen	Dirección origen	Puerto origen
	Dirección destino	Dirección destino	Puerto destino
	Ethertype	Protocolo IPv4	
	VLAN ID	ToS	
	Prioridad VLAN		

Tabla 2-1: Campos de coincidencia de flujos

Sobre los flujos que se caracterizan por los parámetros anteriores el switch OpenFlow puede realizar una de las siguientes acciones:

- **Enviar,** el paquete por los puertos físicos y una serie de puertos virtuales predefinidos como son ALL, CONTROLLER o IN_PORT, el primer puerto envía los paquetes por todas las interfaces excepto la de entrada, el segundo envía el paquete al controlador y el tercero envía el paquete por el puerto de entrada. También se puede especificar un único puerto de salida.
- **Descartar,** el paquete.
- **Encolarlo**, asignarlo a una cola determinada del puerto.
- **Modificar un campo,** como pueden ser, por ejemplo, las direcciones MAC, las direcciones IP o los puertos TCP/UDP origen y destino.
- **Añadir cabeceras,** VLAN (Virtual Local Area Network), MPLS (Multi-Protocol Label Switching) o PBB (Provider Backbone Bridge).
- **Cambiar el TTL,** esta acción permite modificar el valor del TTL (Time to Life) de IPv4, el "Hop Limit" de IPv6 o el TTL de MPLS.

2.4 MININET

Mininet [18] es un emulador de red creado con la finalidad de prototipar rápidamente grandes redes con los recursos de un solo ordenador. Mininet proporciona una API en Python extensible para programar topologías y levantar escenarios. Como alternativa a Mininet existen varios simuladores, como ns-2 u Opnet, pero estos son poco realistas, ya que el código creado en el simulador no será el mismo que en una red real.

Como alternativas más realistas a los simuladores también se pueden emplear máquinas virtuales. El inconveniente de los sistemas basados en máquinas virtuales, como GNS3 es la cantidad de memoria que emplean ya que esto limita el número de elementos que pueden formar la red virtual.

Mininet crea host virtuales, utilizando un método de virtualización basado en procesos y el mecanismo de "network namespaces" [19]. Mininet virtualiza lo "mínimo necesario"[3], aislando básicamente los elementos de red y utilizando network namespaces para aislar y diferenciar el tráfico que se envía entre los hosts de la red virtual generada. Todos los hosts comparten el mismo sistema de ficheros global. Debido a esto se pueden considerar los contenedores de Mininet igual o más ligeros, que otros como pueden ser LXC o Docker, aunque ofrecen menos aislamiento.

En todos los casos, los contendores suponen una menor sobrecargan respecto a la que generan las máquinas virtuales, ya que un sistema operativo completo se ejecuta sobre este hardware virtual, en cada instancia de una máquina virtual. Al utilizar contenedores, que son

[3] Como explica Bob Lantz, en esta lista de correos sobre Mininet de la universidad de Standford https://mailman.stanford.edu/pipermail/mininet-discuss/2014-July/004763.html],

más ligeros, Mininet permite crear redes virtuales con un mayor número de elementos, respectos a otros emuladores que emplean técnicas de virtualización más pesadas.

Por otro lado, Mininet emplea por defecto switches virtuales que soportan OpenFlow. Estos switches llamados Open vSwitch [20], son switches software open source licenciados bajo la licencia Apache 2.0 que están diseñados para trabajar con el protocolo OpenFlow además de soportar interfaces de gestión y protocolos estándar, de monitorización como pueden ser NetFlow, sFlow, IPFIX o RSPAN. Open vSwitch se explica con más detalle en la sección 2.6.

En Mininet los enlaces entre los distintos elementos de la red virtual, son implementados usando un driver virtual Ethernet "veth". Los dos extremos de un enlace constituyen un "veth pair". El par veth se utiliza en Linux para interconectar los diferentes namespaces. En la red creada con Mininet, los paquetes que se envían entre hosts son procesados en memoria por la pila de red real del kernel de Linux sin hacer uso de dispositivos de red físicos.

La diferencia entre Mininet y otros emuladores ligeros como CORE, IMUNES o VNX es que Mininet fue uno de los primeros en centrarse en el prototipado de SDN, incluyendo la posibilidad de usar OpenFlow.

Como se explica en [21], uno de los principales inconvenientes de Mininet es la falta de confiabilidad en el rendimiento, especialmente con grandes cargas. Los recursos de CPU son multiplexados en el tiempo por el planificador (scheduler) de Linux. Esto no garantiza que un host que está listo para enviar un paquete sea puesto en ejecución inmediatamente, o que todos los switches vayan a enviar a la misma tasa. Como todos los elementos de la red virtual comparten los mismos recursos hardware, realizar experimentos a gran escala puede ser complicado ya que el sistema físico puede entrar en saturación modificando el comportamiento estándar.

2.5 DOCKER

Docker [22], es un proyecto Open Source que proporciona una manera sistemática de automatizar y desplegar de manera rápida aplicaciones Linux utilizando un contenedor portable. Docker, como explica [9], se ha convertido rápidamente en un estándar de facto entre las herramientas de gestión y formato de imagen para contenedores.

Docker hace uso de LXC [23], que es una herramienta para crear y gestionar fácilmente contenedores. Sobre esta herramienta Docker proporciona una API a nivel de kernel y aplicación que permite ejecutar procesos con aislamiento de CPU, memoria, entrada/salida, red, etc. Los contenedores Docker se crean utilizando imágenes base. Una imagen Docker puede incluir solo el sistema operativo fundamental o pude consistir en una aplicación sofisticada lista para ejecutarse.

Una de las características clave de Docker son las imágenes del sistema de ficheros por capas. Como vemos en la Figura 2-6, un contenedor tendrá una capa que puede escribir y modificar y una capa de imagen subyacente. Las capas de imagen se reutilizan entre contenedores reduciendo el uso de espacio en disco y simplificando la gestión del sistema de ficheros. Una sola imagen de SO se puede usar como base para muchos contenedores mientas que cada contenedor puede tener su propia capa de ficheros modificados. Cuando se elimina el contenedor también se elimina la capa "escribible" eliminando todos los datos que se hayan generado en el contenedor y que no se hayan escrito en un "data volume". Un "data volume" es un directorio o fichero en el sistema de ficheros anfitrión que se monta directamente en un contenedor, por lo que varios contenedores pueden compartir un "data volumen".

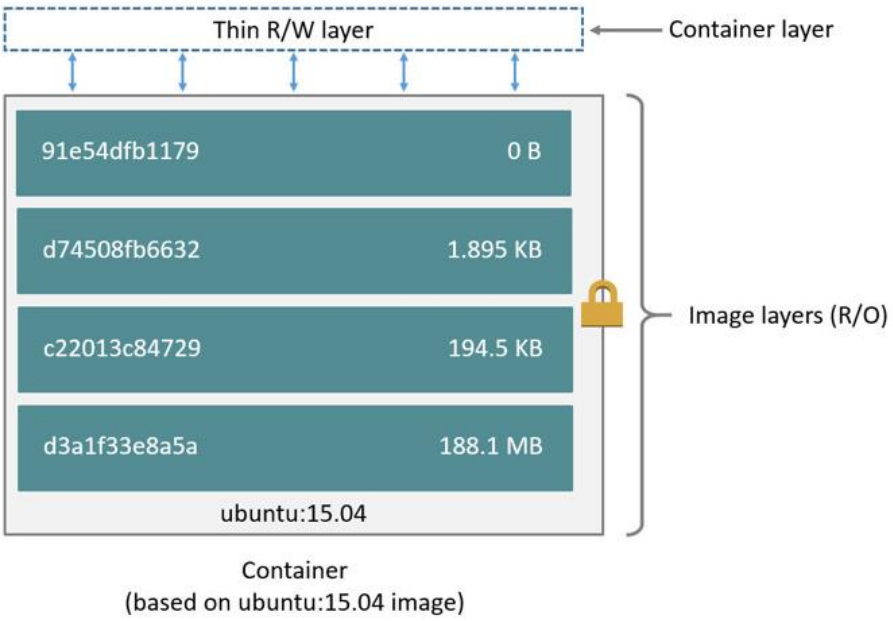

Figura 2-6: Capas contenedor Docker [24]

2.6 OPEN VSWITCH

Open vSwitch [25] es un switch software creado para su uso en entornos virtuales. Open vSwitch se diferencia de las aproximaciones tradicionales en que exporta una interfaz para control del tráfico que puede usarse para aplicar QoS, tunneling y establecer reglas de filtrado con granularidad fina.

La implementación de Open vSwitch consiste en dos componentes: a nivel de kernel el "fast path" y en espacio de usuario el "slow path". El "slow path" se trata principalmente del demonio ovs-vswitchd que supone la mayor parte de la funcionalidad de Open vSwitch. El "fast path" implementa el motor de reenvío, que es responsable de la búsqueda, modificación y reenvío de paquetes.

El datapath del kernel que procesa paquetes con un sistema basado en reglas, mantiene una tabla de flujos en memoria que asocia flujos activos con acciones a realizar. Un paquete que tiene una coincidencia en la tabla de flujos es procesado exclusivamente por el "fast path". Si no hay coincidencia, se pasará ese paquete a espacio de usuario para decidir qué hacer con él. Este mecanismo es similar al del protocolo OpenFlow y la relación del switch con el controlador. El "fast path" implementa solo las funcionalidades críticas respecto a velocidad debido a que el kernel es específico de cada sistema. Esta arquitectura permite que se pueda migrar a otras plataformas no-Linux o implementarlo mediante hardware para aumentar rendimiento [26].

La estructura de la base de datos de un demonio Open vSwitch (ovs-vswitchd), se muestra en la Figura 2-7. El programa *ovs-vsctl* [27] proporciona una interfaz para la configuración de la base de datos, este programa permite, entre otras cosas:

- Crear un nuevo bridge: *ovs-vsctl add-br <bridge>*
- Añadir un puerto a un bridge: *ovs-vsctl add-port <bridge> <port>*
- Asignar al bridge un controlador: *ovs-vsctl set-controller <bridge> tcp:<ip>[:<port>]*

- Listar los bridges instanciados: *ovs-vsctl lis-br o ovs-vsctl show*
- Eliminar un bridge: *ovs-vsctl del-br <bridge>*
- Mostrar alguna tabla de la base de datos: *ovs-vsctl list <table>*

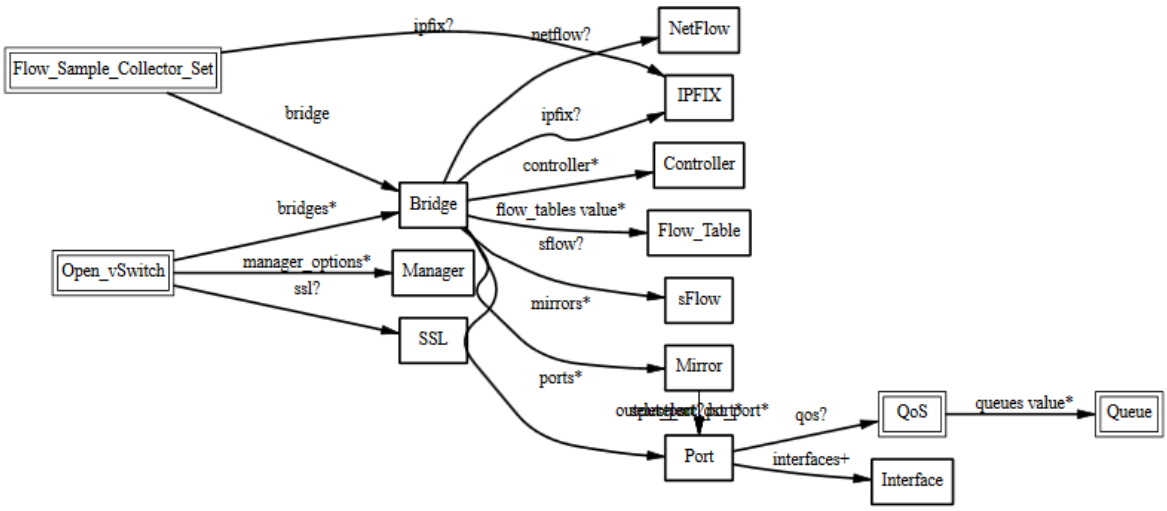

Figura 2-7: Base de datos Open vSwitch [28]

La configuración de alto nivel para el demonio es la tabla "Open_vSwitch", que tiene que tener exactamente una entrada. El demonio Open vSwitch puede tener cero o más bridges, que registran su configuración en la tabla Bridge.

Un bridge es una pieza de software utilizada para unir dos o más segmentos de red. A efectos prácticos un bridge se comporta como un switch virtual, al que se puede conectar cualquier dispositivo real (ej. eth0) o virtual (ej: tap0).

A su vez el bridge o switch virtual puede tener cero o maś puertos. Al crear un switch con el comando *ovs-vsctl add-br* el switch se crea sin puertos adicionales (aparte de su propio puerto), como se ve en el ejemplo de la Figura 2-8.

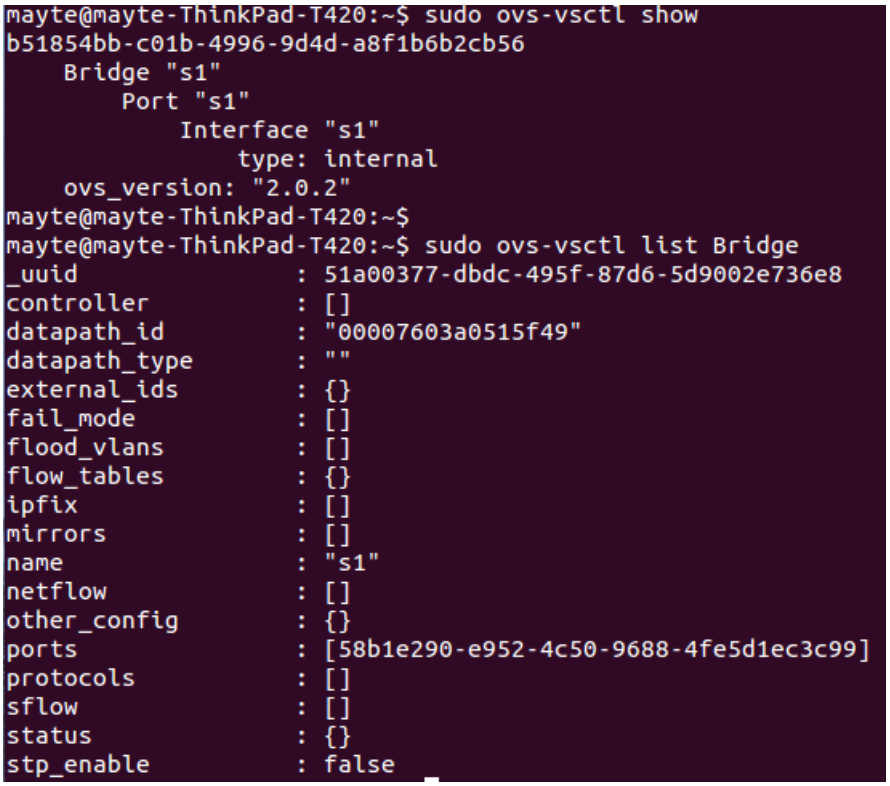

Figura 2-8: Vista de tabla Bridge por defecto

Cada puerto del switch virtual tiene al menos una interfaz y vemos que se pueden aplicar a los puertos políticas de calidad de servicio QoS.

Por otro lado, el switch virtual puede realizar port mirroring y enviar ciertas tramas a puertos "espejo", además de a su destino real. Además, soporta los protocolos de monitorización sFlow, NetFLow e IPFIX.

En el esquema de la base de datos, vemos que se pueden añadir configuraciones a las tablas de flujos OpenFlow. Por ejemplo, por cada tabla se puede limitar el número de flujos, lo que puede ser de utilidad en caso de tener restricciones de rendimiento o almacenamiento.

Un bridge de Open vSwitch puede tener uno o más controladores, las configuraciones por defecto de los controladores básicamente especifican la IP y el puerto donde se encuentra escuchando el controlador.

2.7 TRABAJO RELACIONADO

ClickOS [29] es una plataforma basada en Xen y Click, para la instanciación de middleboxes en hardware no especializado con la finalidad de proporcionar un alto rendimiento. Click es un router software modular que emplean para desarrollar los middleboxes software. Por otro lado, se basan en Xen para desarrollar un entorno de virtualización más eficiente, reduciendo consumo de memoria y tiempo de arranque. Este trabajo se centra en el rendimiento y reusabilidad de las funciones virtuales de red no atendiendo a una solución de NFV completa considerando la red en la que se integrarán las funciones de red.

ClickOS que se basa en métodos de virtualización de funciones de red basados en Hypervisores, que resultan en peor rendimiento, son más pesadas y requieren de un mayor tiempo a la hora de la creación y eliminación de las funciones virtuales a diferencias de sistemas como el nuestro basados en contenedores, además dicho trabajo no está pensado

inicialmente para su integración en redes definidas por software.

GLANF [30] se trata de un entorno abierto para crear, desplegar y gestionar funciones virtuales de red (NFV) en redes con OpenFlow habilitado. Se emplean funciones de red basadas en contenedores Docker para conseguir baja sobrecarga del rendimiento, rápido despliegue y alta reusabilidad que falta en las implementaciones de NFV actuales. A través de la API northbound de SDN las funciones de red se pueden instanciar, el tráfico se puede encaminar con la política de encadenado deseada y las aplicaciones pueden recibir notificaciones. GLANF es transparente, los hosts no tienen que cambiar el destino del tráfico para usar las funciones de red. El sistema contiene las funciones de red en un servidor, que denomina "servidor GLANF" y redirige el tráfico que lo requiera a cada función de red, conectada a través de Open vSwtich. La redirección del tráfico u orquestación se gestiona a través de un controlador SDN OpenDayLight.

A diferencia del primer trabajo descrito, GLANF sí que emplea contenedores y está pensados para una integración en una red SDN pero en la arquitectura de GLANF se plantea la ubicación de las funciones virtuales de red dentro de un servidor concreto llamado "servidor GLANF", teniendo que redirigir el tráfico al mismo. Esta característica hace que se pueda añadir retardo en el tráfico lo cual no es deseable. Por otro lado, nuestro sistema plantea la posibilidad de instanciar las funciones virtuales de red allá donde sean necesarias siempre que se tenga acceso a un equipo con Open vSwitch instalado y que se permita la virtualización basada en contenedores. En caso de que esto no sea posible, nuestro sistema puede ejecutarse en un servidor de la misma manera que lo hace GLANF.

En [31] se presenta un método para interconectar fácilmente contenedores que se ejecutan en equipos diferentes con el fin de permitir el aprovisionamiento bajo demanda de contenedores Linux usando redes definidas por software. Los contenedores que utiliza son Docker, lo contenedores se conectan a la red mediante switches virtuales Open vSwitch gestionados por un mismo controlador OpenDaylight.

El último trabajo citado no especifica la finalidad de la aplicación de los contenedores, pero proporciona un método para el aprovisionamiento bajo demanda de los mismos en una red SDN, sin embargo, no hace ningún tipo de evaluación de rendimiento o funcionalidad sobre el método. En nuestro trabajo se detalla el despliegue de contenedores bajo demanda, con la funcionalidad especifica de funciones virtuales de red y se evalúa el rendimiento del sistema con métricas del ancho de banda que se alcanza.

3 DISEÑO Y DESARROLLO

En este trabajo de fin de master se ha realizado, el desarrollo de funciones virtuales de red para monitorización embebidas en el propio equipo de red para entornos SDN. Las sondas desarrolladas deben tener la capacidad de moverse transparentemente entre los equipos de comunicaciones y los equipos utilizados como sondas de red tradicionales.

Concretamente se han desarrollado dos funciones de red, una de ellas tiene la finalidad de monitorizar el tráfico HTTP y la segunda función de red tiene la finalidad de detectar un posible ataque de denegación de servicio basado en SYN flooding. Ambas funciones de red se han desarrollado en lenguaje C. En los siguientes apartados se detalla el diseño y desarrollo de las mismas. Posteriormente en esta misma sección se describe el método ideado para permitir la virtualización de dichas funciones en un entorno SDN y su movimiento entre diferentes entornos de monitorización.

3.1 NFV MONITORIZACIÓN HTTP

El Protocolo de Transferenica de Hypertexto (HTTP por sus siglas en inglés), es un protocolo a nivel de aplicación para sistemas de información hypermedia distribuidos y colaborativos [32]. El protocolo HTTP se emplea en la World-Wide Web desde 1990 y su uso es muy extendido en la actualidad.

Como datos a tener en cuenta sobre el protocolo, hay que decir que se trata de un protocolo sin estado, por lo que ni el emisor ni el receptor, guardan ningún dato y el emisor no espera ningún mensaje de confirmación (ACK) por parte del receptor. Otra característica es que se trata de un protocolo basado en petición-respuesta, de modo que, el cliente hace peticiones al servidor y el servidor responde al cliente. De forma breve la petición comienza con un método de petición, la URI, y la versión del protocolo seguido de la secuencia "\r\n" (CRLF. La respuesta contiene una línea de estado, incluyendo versión del protocolo del mensaje, un código de éxito o error y una frase explicativa del código seguido de la secuencia "\r\n" (CRLF). Los campos que se han citado, tanto en la línea de petición como en la de estado, se separan por uno o más espacios simples (SP).

Los métodos de petición de HTTP pueden ser OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE, CONNECT, pero en esta aplicación solo se analizan los métodos GET y POST debido a su relevancia y uso a día de hoy. Estos dos métodos se emplean para solicitar la recepción de ciertos datos y la subida de datos respectivamente. A continuación, en las Tabla 3-1 y Tabla 3-2, se muestran ejemplos de peticiones GET y POST y respuestas.

Método	**SP**	**URI**	**SP**	**Versión**	**CRLF**
GET	SP	/index.html	SP	HTTP/1.1	CRLF
POST	SP	/inedex.html	SP	HTTP/1.0	CRLF

Tabla 3-1: Línea de petición HTTP

Versión	**SP**	**Código de estado**	**SP**	**Razón**	**CRLF**
HTTP/1.1	SP	200	SP	OK	CRLF
HTTP/1.1	SP	404	SP	Not Found	CRLF

Tabla 3-2: Línea de estado HTTP

La finalidad de la función de red es monitorizar tráfico HTTP e imprimir información relevante como el TimeStamp de la petición, IP origen, IP destino, método y URI, emparejando las peticiones HTTP (GET o POST) con sus respuestas. Para ello, de cada paquete, que se recibe, se extraen y guardan en una estructura los siguientes datos:

TimeStamp	IP origen	IP destino	Puerto origen	Puerto destino	Método	URI	protocolo

Tabla 3-3: Datos que se extraen en monitor HTTP

A continuación, en la Figura 3-1 ,se muestra el diagrama de flujo del programa

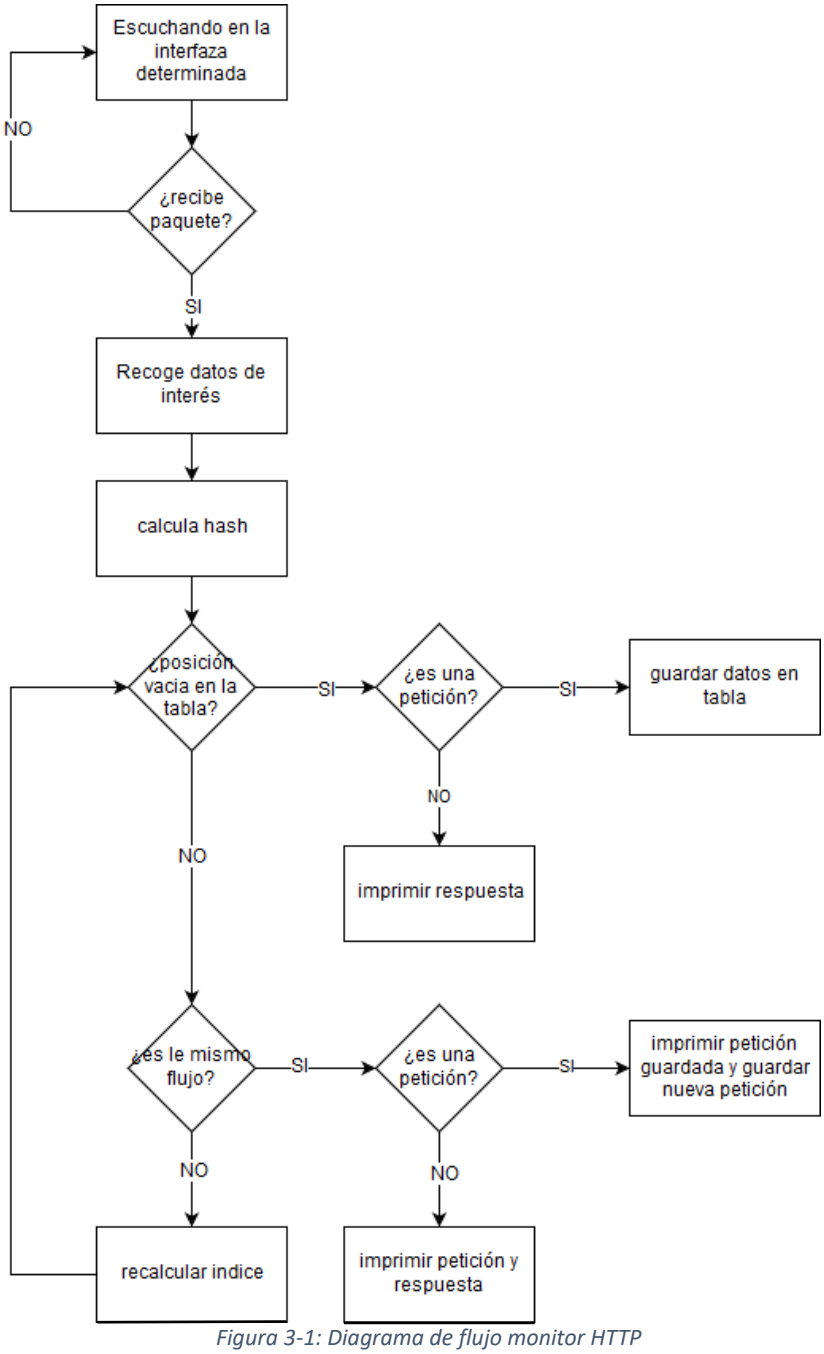

Figura 3-1: Diagrama de flujo monitor HTTP

El hash se calcula a partir de las direcciones IP origen (IPsrc) y destino (IPdst), los puertos origen (PortSrc) y destino (PortDst) y el protocolo (proto) del siguiente modo:

y = (IPsrc + IPdst + PortSrc + PortDst + proto)% tamañoTabla

En caso de colisión se recalcula el hash del siguiente modo:

Pertub = y
perturb >> 5
*y = (5 * y) + 1 + perturb*
y = y % tamaño_tabla

Para comprobar si es el mismo flujo se comparan los campos que se emplean para calcular el hash.

El fichero de salida de la función de monitorización mostrará los datos de las peticiones y respuestas con los campos que se muestran en la Tabla 3-4 separados por un tabulador (\t). Hay que tener en cuenta que los campos que componen último campo, la línea de petición HTTP o de estado según sea una petición o una respuesta, están separados por espacios, del mismo modo en que se recibe.

Time Stamp	IP origen	IP destino	Línea petición/estado HTTP

Tabla 3-4: Formato de salida fichero monitor HTTP

De modo que, el aspecto que muestra el fichero de salida es el siguiente, con cada par de pregunta respuesta separado por una línea.

```
------------------------------------------------------------

1361916156.044817     254.16.133.132 98.138.19.88     HTTP/1.1 200 OK

------------------------------------------------------------

1361916156.050428     254.16.133.28   172.16.139.250 GET
/CSIS/CSISISAPI.dll/?request?b2bc13b2-c435-4845-8c7d-
99228306a0aa;CCSISSvrCFG%3A%3AgetWorkgroupAgentStatus%3B4%3B4240 HTTP/1.1

------------------------------------------------------------

1361916156.083885     254.16.133.99   172.16.139.250 GET
/CSIS/CSISISAPI.dll/?request?da313d4f-e9ca-4de1-a01f-
0047540fbfbc;CCSISSvrCFG%3A%3AgetWorkgroupAgentStatus%3B4%3B4217 HTTP/1.1

------------------------------------------------------------
```

Tabla 3-5: Fichero de salida monitor HTTP

En el Anexo I se puede consultar el código de este programa.

3.2 NFV MONITOR DE SYN FLOODING

Un ataque de inundación de SYN (SYN flooding) es un tipo de ataque de denegación de servicio (DoS). En este ataque se envía un gran número de segmentos SYN TCP, sin completar el handshake en tres fases de TCP. Cuando la tasa de paquetes SYN generados es muy alta, el servidor pasa más tiempo respondiendo a paquetes SYN (y reservando recursos para las conexiones) que atendiendo peticiones legítimas lo cual provoca una caída en el servicio.

Esta función de red pretende generar un fichero de log sobre ataques de denegación de servicio tipo SYN flooding. Para ello se mide la tasa de SYN/seg que llega a cada IP destino desde múltiples IP origen. Se mide la tasa contando el número de segmentos SYN que llegan a una IP destino desde distintas IP orígenes y se comprueba dicho contador en intervalos de tiempo regulares determinados por un parámetro configurable. En el fichero de log se imprimen las direcciones IP destino cuya tasa de SYN/seg supere un umbral configurable en el programa.

El programa hace uso de dos hilos de proceso. En el hilo principal del programa, se van leyendo los paquetes que llegan a la interfaz de red en la que se está escuchando. De los paquetes que se van recibiendo, se extraen los parámetros necesarios para el análisis: IP destino, Puerto destino, IP origen, Puerto origen y valor del bit SYN.

Si el bit SYN está activado se guardan esos datos en una tabla hash de direccionamiento abierto indexada por IP destino. El hash de la tabla se calcula a partir de la IP del siguiente modo, dada una dirección IP W.X.Y.Z y una tabla de tamaño T posiciones.

$$ind = (W+X+Y+Z) \% T$$

En caso de colisiones se recalcula la posición en la que guardar los datos del siguiente modo.

```
perturb = ind;
ind = (5*ind)+1+perturb;
perturb >> 5;
ind = ind % T;
```

Siendo *ind* la posición originalmente calculada por la función hash.

Los datos en la tabla se guardan del modo que se observa en la Figura 3-2.

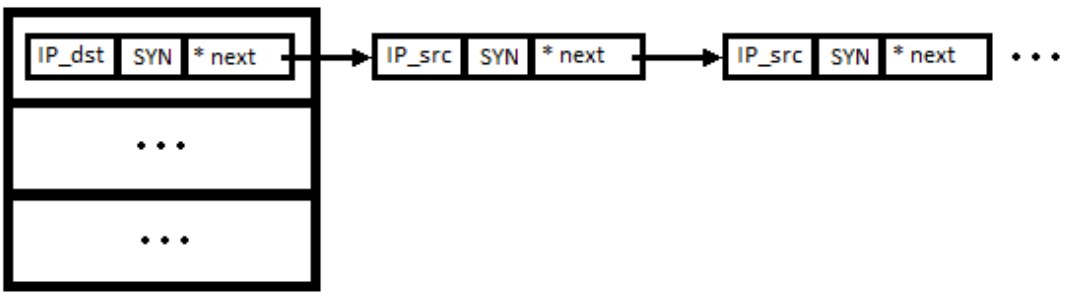

Figura 3-2: Estructura de datos monitor syn flooding

Cada entrada de la tabla guarda una estructura que contiene la IP destino, el número de segmentos SYN que ha recibido y un puntero al siguiente nodo de una lista enlazada. Los nodos de la lista enlazada contienen cada IP origen que ha enviado un segmento SYN a dicha IP destino y el número de segmentos SYN que ha enviado esa IP origen concreta.

Concurrentemente, el segundo hilo o hilo de exportación, comprueba dicha tabla, en intervalos de tiempo determinado por un parámetro configurable. En este hilo se recorre la tabla y se imprimen en un fichero las entradas que superen un umbral de SYN/seg establecido por otro parámetro configurable. De modo que si, por ejemplo, se establece una velocidad para el hilo de 10 segundos y el umbral está en una tasa de 4 SYN/seg, cuando se ejecute el umbral se imprimirán las entradas de la tabla que tengan un número de SYN mayor a 40.

El diagrama de flujo del hilo principal se muestra en la Figura 3-3 y en la Figura 3-4 se muestra el diagrama de flujo del hilo de exportación.

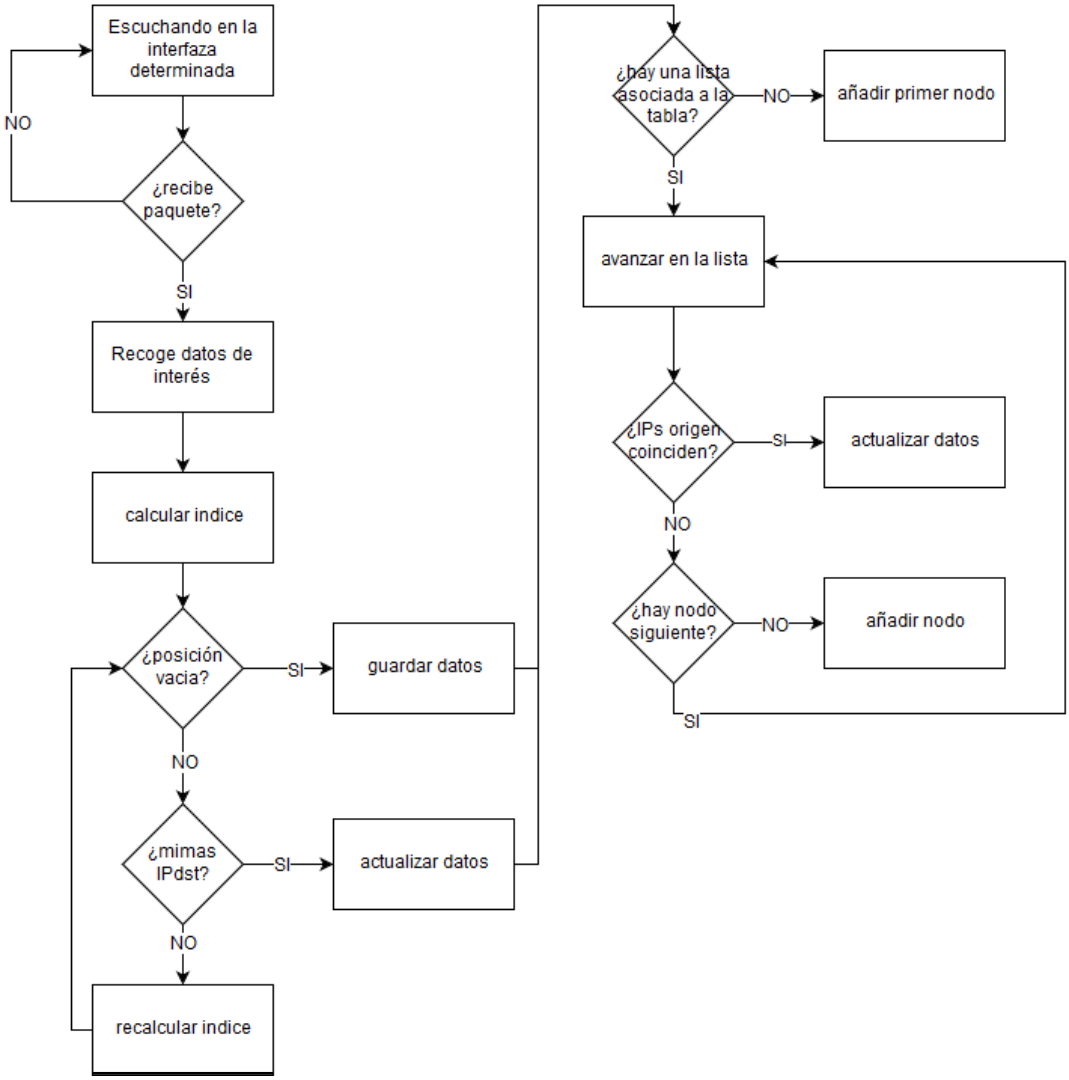

Figura 3-3: Diagrama de flujo hilo principal monitor SYN flooding

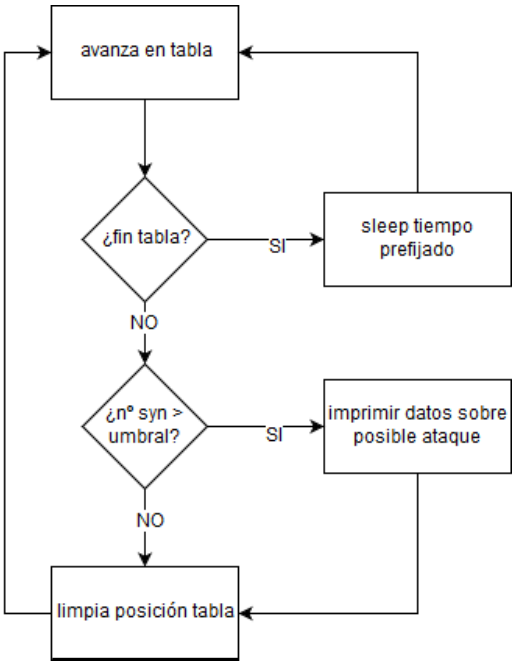

Figura 3-4: Diagrama de flujo hilo exportación SYN flooding

Los datos se imprimen en un fichero de salida tipo .csv, separando los datos por comas. Los datos que se imprimen son:

1. Timestamp
2. IP destino
3. número de syn que ha recibido
4. IP origen
5. número de syn generados por dicha IP origen.

Los campos 4 y 5, de la lista anterior, se repiten tantas veces como número de IPs hayan enviado segmentos a la IP destino, con el bit SYN activo. De modo que un ejemplo del fichero se ve en la Tabla 3-6 y el código del programa en el Anexo II

1496244432.284590,172.16.133.82,10,96.43.146.176,1,74.125.226.222,1,216.137.33.209,5,64.56.203.22,3,
1496244432.284590,172.16.133.132,10,98.139.134.187,2,76.13.6.174,2,98.139.161.29,2,98.138.19.88,2,98.139.240.23,2,
1496244432.284590,172.16.139.250,71,172.16.133.13,2,172.16.133.20,2,172.16.133.28,2,172.16.133.99,2,172.16.133.87,2,172.16.133.116,6,172.16.133.35,2,172.16.133.84,2,172.16.133.49,2,172.16.133.95,2,172.16.133.92,2,172.16.133.29,2,172.16.133.44,2,172.16.133.207,2,172.16.133.83,2,172.16.133.93,2,172.16.133.11,2,172.16.133.55,2,172.16.133.73,2,172.16.133.53,2,172.16.133.21,2,172.16.133.30,2,172.16.133.34,2,172.16.133.25,2,172.16.133.97,2,172.16.133.33,2,172.16.133.68,2,172.16.133.12,2,172.16.133.63,2,172.16.133.66,2,172.16.133.82,2,172.16.133.37,2,172.16.133.78,2,172.16.133.67,1,

Tabla 3-6: Fichero de salida monitor syn flooding

3.3 VIRTUALIZACIÓN DE LAS FUNCIONES DE RED

Se han considerado dos tecnologías para llevar a cabo esta tarea. Por un lado, se ha planteado la utilización de Mininet para hospedar las funciones virtuales de red y la comunicación con el resto de la red. Por otro lado, para poder cumplir el objetivo de la portabilidad de las funciones de red entre distintos equipos se ha optado por la utilización de Docker y Open vSwitch. De esta manera se podrá comparar la eficiencia de las tecnologías, así como su flexibilidad y posibilidades.

En ambos casos se parte de un esquema inicial como el de la Figura 3-5. Este esquema contiene una única función virtual. En caso de implementarse con Mininet la instanciación del Open vSwitch y de la máquina virtual o contenedor que implementa la función virtual se realiza a través de la propia herramienta Mininet, mientras que para el otro caso inicialmente se hizo de manera manual mediante comandos en la terminal de Linux y luego se automatizó con un script.

Una vez se validó el esquema inicial, se desarrollaron dos pruebas de concepto con las funciones virtuales anteriormente descritas: monitorización del protocolo HTTP y detección de ataques de denegación de servicio (DoS, Denial Of Service) basados en SYN flooding. La finalidad de usar funciones de monitorización desagregadas es poder realizar modelos más complejos en los que se monitorice simultáneamente distintos tipos de tráfico de manera paralela.

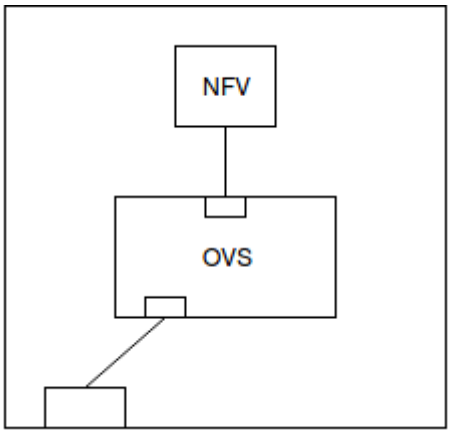

Figura 3-5: Esquema inicial

Para completar la virtualización de las funciones de red se plantea, un mecanismo para mover las sondas de monitorización entre distintos equipos físicos. De este modo se podrá mover la monitorización bajo demanda a donde haga falta y de una manera transparente para el usuario.

A continuación, en los distintos apartados que siguen se concreta y desglosa el conjunto de los elementos desarrollados.

3.3.1 Topología sobre Mininet

Primeramente, se empleará Mininet para instanciar una topología con tantos hosts como funciones de red se quiera disponer en el equipo, ya sea mediante un script o utilizando el entorno gráfico "MiniEdit". Una vez Mininet se esté ejecutando con un switch instanciado, se conectará a la interfaz física del equipo anfitrión.

Para implementar el esquema inicial, se generó una topología en Mininet con un host y un switch OpenFlow. En el siguiente fragmento de código, se muestra como siendo s1 el switch, este se conecta a la interfaz real eth0. En primer lugar, a s1 se le añade como puerto la interfaz real eth0, luego se deshabilita esta interfaz y finalmente se configura s1 con la IP del equipo y la puerta de enlace predeterminada. De este modo se recibirá en el switch OpenFlow el tráfico de la interfaz eth0.

```
ovs-vsctl add-port s1 eth0

ifconfig eth0 0

ifconfig s1 192.168.1.220 netmask 255.255.255.0

route add default gw 192.168.1.1 s1
```

Tabla 3-7: Conectar OVS a interfaz real

Una vez ejecutados los comandos anteriores en el servidor que albergará las funciones de red, comprobamos que el switch OpenFlow tiene un puerto conectado a la interfaz eth0 mediante el comando ***ovs-vsctl show*** y que la interfaz de la puerta de enlace es s1 mediante el comando ***route***, como se puede ver en la Figura 3-6.

```
mayte@virtual-machine:~/Documentos$ sudo ovs-vsctl show
9f3ad3be-1712-4f9f-bb40-fd33f307d08d
    Bridge "s1"
        fail_mode: secure
        Port "s1"
            Interface "s1"
                type: internal
        Port "eth0"
            Interface "eth0"
        Port "s1-eth1"
            Interface "s1-eth1"
    ovs_version: "2.0.2"
mayte@virtual-machine:~/Documentos$ route
Tabla de rutas IP del núcleo
Destino         Pasarela        Genmask         Indic Métric Ref    Uso Interfaz
default         192.168.1.1     0.0.0.0         UG    0      0        0 s1
192.168.1.0     *               255.255.255.0   U     0      0        0 s1
```

Figura 3-6: ovs-vsctl show & route

A continuación, se configura el switch Openflow para que reenvíe el tráfico que proceda a las máquinas virtuales. Esto se puede hacer añadiendo un controlador SDN local en el mismo servidor, lo que da más flexibilidad permitiendo actuar sobre el tráfico de manera dinámica. Otra opción consiste en instalar la regla manualmente en el switch y que actúe sobre el tráfico siempre del mismo modo de manera estática.

En esta primera aproximación instalamos una regla en el switch para que reenvíe todo el tráfico que le entre por el puerto eth0 al puerto s1-eth1.

Para ver la numeración que tienen los puertos del switch, se ejecuta el siguiente comando [33].

```
sudo ovs-ofctl dump-ports-desc s1
```

Tabla 3-8: Ver detalle de puertos asociados a OVS

Cuya ejecución devolverá un resultado similar al siguiente:

```
mayte@virtual-machine:~/Documentos$ sudo ovs-ofctl dump-ports-desc s1
OFPST_PORT_DESC reply (xid=0x2):
 1(s1-eth1): addr:5a:70:ee:a3:0a:16
     config:     0
     state:      0
     current:    10GB-FD COPPER
     speed: 10000 Mbps now, 0 Mbps max
 2(eth0): addr:08:00:27:7f:05:f6
     config:     0
     state:      0
     speed: 0 Mbps now, 0 Mbps max
 LOCAL(s1): addr:08:00:27:7f:05:f6
     config:     0
     state:      0
     speed: 0 Mbps now, 0 Mbps max
```

Figura 3-7: ovs-ofctl dump-ports-desc

Conociendo los puertos se instala la siguiente regla en el switch, para que todo el tráfico que entre en el equipo se reenvíe al host que alberga la función virtual de red y a continuación se comprueba que se ha instalado la regla en el switch como se ve en la Figura 3-8.

```
sudo ovs-ofctl s1 idle_timeout=3600,in_port=2,actions=output:1

sudo ovs-ofctl dump-flows s1
```

Tabla 3-9: Instalar y mostrar reglas en tabla de flujos del OVS

La ejecución de estas instrucciones devolverá el siguiente resultado:

```
mayte@virtual-machine:~$ sudo ovs-ofctl add-flow s1 idle_timeout=3600,\
> in_port=2,actions=output:1
mayte@virtual-machine:~$ sudo ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=10.158s, table=0, n_packets=0, n_bytes=0, idle_timeo
ut=3600, idle_age=10, in_port=2 actions=output:1
```

Figura 3-8: ovs-ofctl add-flow & ovs-ofctl dump-flows

Una vez hecho esto tenemos el escenario configurado.

Puesto que inicialmente el escenario de pruebas se realiza con un único equipo, se crea una máquina virtual VirtualBox con Ubuntu y se instala Mininet. Para poder enviar tráfico a la máquina virtual y comprobar que el contenedor de Mininet recibe el tráfico se conecta el adaptador de red de la máquina virtual a una interfaz TAP en el ordenador anfitrión. Como se ve en Figura 3-9.

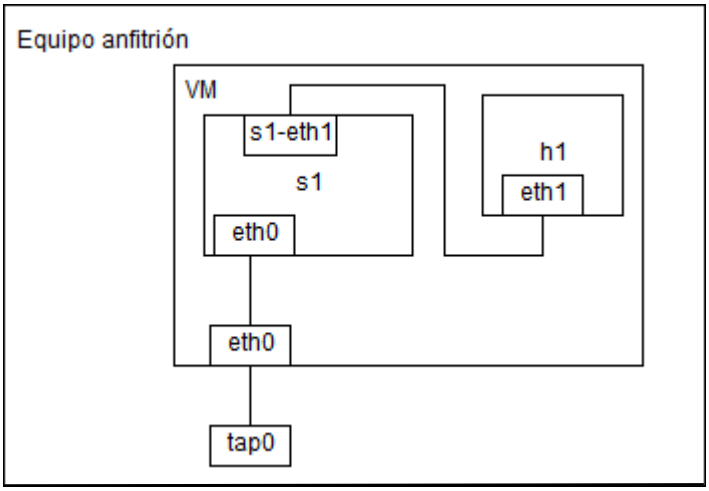

Figura 3-9: Escenario Mininet

Las interfaces TAP son interfaces software, ya que solo existen en el kernel del sistema operativo y, a diferencia de las interfaces de red comunes, no tienen un componente hardware físico asociado. Cuando se envían datos a través de una interfaz TAP, el kernel envía los datos a un programa en el espacio de usuario que está conectado a la interfaz.

Para crear una interfaz TAP por defecto y levantar dicha interfaz se ejecutan los siguientes comandos:

```
ip tuntap add mode tap tap0

ip link set tap0 up
```

Tabla 3-10: Crear interfaz software TAP

El resultado de la ejecución deberá ser el que se muestra en la Figura 3-10.

```
mayte@virtual-machine:~$ sudo ip tuntap add mode tap tap0
mayte@virtual-machine:~$ sudo ip link set tap0 up
mayte@virtual-machine:~$ ifconfig tap0
tap0      Link encap:Ethernet  direcciónHW 66:b1:a7:5f:2e:eb
          ACTIVO DIFUSIÓN MULTICAST  MTU:1500  Métrica:1
          Paquetes RX:0 errores:0 perdidos:0 overruns:0 frame:0
          Paquetes TX:0 errores:0 perdidos:0 overruns:0 carrier:0
          colisiones:0 long.colaTX:500
          Bytes RX:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Figura 3-10: tunctl & ip link list

Por último, se envía tráfico a la interfaz creada y se comprueba que efectivamente se está recibiendo el tráfico en el host.

3.3.2 Topología sobre Docker

En este escenario, el host que albergará la función de monitorización se genera mediante Docker y se instancia el switch virtual Open vSwitch manualmente. El escenario inicial que se crea es el mismo que ya se ha descrito en el apartado 3.1, con un host que albergará la función virtual y un switch conectado a la interfaz real del equipo.

Por un lado, se crea el host Docker, mediante un Dockerfile; un Dockerfile es un documento de texto con las líneas de comando para ensamblar una imagen de una máquina virtual. Como vemos a continuación se crea el contenedor docker, con Ubuntu, se copia en la carpeta /tmp/ del contenedor el ejecutable de la función de red que se nombra "bw" y se instalan varios programas que pueden resultar útiles como son libpcap, iperf, netcat y tcpdump.

```
FROM ubuntu
COPY bw /tmp/

RUN apt-get update
RUN apt-get install -y libpcap-dev
RUN apt-get install -y iperf
RUN apt-get install -y netcat
RUN apt-get install -y tcpdump
```

En el mismo directorio donde está el fichero Dockerfile y el ejecutable "bw" se ejecutan los siguientes comandos para crear el contenedor a partir del Dockerfile que se ha llamado "bwmes". Luego, una vez creado, se ejecuta el contendor y se inicia sin interfaces de red (--net='none'), para evitar que se cree la red virtual por defecto de Docker en la que están conectados todos los contenedores que se ejecuten en el equipo. Por último, se nombra el contenedor como h1. El flag -i mantiene la entrada estándar STDIN abierta y -t crea un pseudo terminal TTY.

```
docker build -t bwmes .

docker run --net='none' -i -t --name h1 bwmes
```

Tabla 3-11: Crear imagen de Dockerfile y crear contenedor

Por otro lado, se crea el switch virtual Open vSwitch y se conecta a la interfaz física, añadiendo la interfaz real a uno de los puertos del switch. Se trata al switch virtual Open vSwitch como la interfaz real asignándole la IP del equipo y estableciendo la puerta de enlace predeterminada.

Las instrucciones necesarias son las siguientes:

```
ovs-vsctl add-br s1

ovs-vsctl add-port s1 eth0

ifconfig eth0 0

ifconfig s1 192.168.1.220 netmask 255.255.255.0

route add default gw 192.168.1.1 s1
```

Tabla 3-12: Conectar OVS a interfaz real

A continuación, se conecta el contenedor Docker al switch virtual a través de un "par veth". Con los dos siguientes comandos, se genera el par veth con dos extremos, que se nombran, par_h y par_o. Y se comprueba que se han generado correctamente viendo la lista de links del equipo, Figura 3-11.

```
ip link add par_h type veth peer name par_o

ip link list
```

Tabla 3-13: Crear par veth

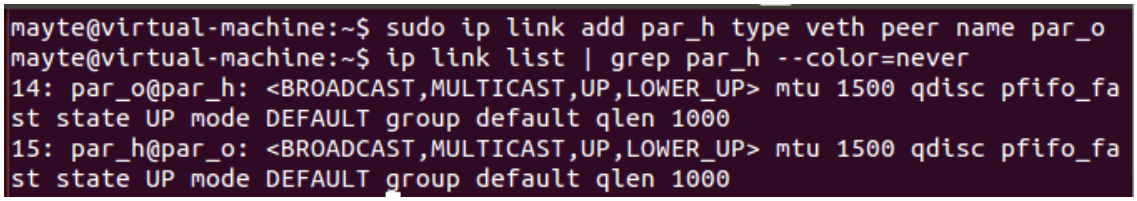

Figura 3-11: ip link list

Recapitulando, en este momento tenemos el switch virtual creado y conectado a la interfaz física de la máquina virtual, el contenedor Docker y el par veth, pero aún no se ha conectado el contenedor con el switch como podemos ver en la Figura 3-12.

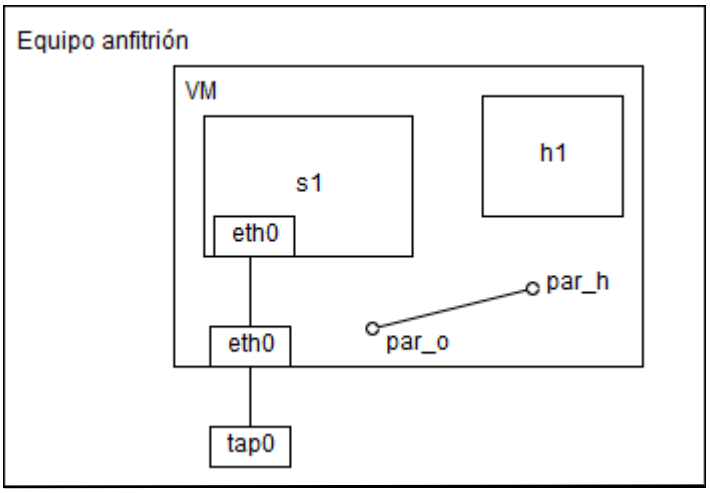

Figura 3-12. Elementos creados, pero no conectados

Se conecta el extremo par_o al Open vSwitch y comprobamos que la acción se haya realizado correctamente, Figura 3-13.

```
ovs-vsctl add-port s1 par_o

ip link set par_o
```

Tabla 3-14: Añadir un extremo del par veth

```
mayte@virtual-machine:~$ sudo ovs-vsctl add-port s1 "par_o"
mayte@virtual-machine:~$ sudo ovs-vsctl show
9f3ad3be-1712-4f9f-bb40-fd33f307d08d
    Bridge "s1"
        Port "s1"
            Interface "s1"
                type: internal
        Port "eth0"
            Interface "eth0"
        Port par_o
            Interface par_o
    ovs_version: "2.0.2"
```

Figura 3-13: ovs-vsctl add-port

A continuación, se mueve el extremo par_h dentro del namespace del contenedor; para ello primero se obtiene el PID del contenedor, que se puede obtener con el comando “inspect” de Docker, que devuelve información de bajo nivel de los objetos Docker.

```
docker inspect -f '{{.State.Pid}}' h1
```

Tabla 3-15: Obtener PID del contenedor

Como se ve en Figura 3-14, el PID que se obtiene es 2815, de modo que continuamos el proceso y se asigna la interfaz par_h del par veth al network namespace del contenedor docker. Las instrucciones para ello son las siguientes:

```
mayte@virtual-machine:~$ sudo docker inspect -f '{{.State.Pid}}' "h1"
2815
```

Figura 3-14: Obtener PID del contenedor

```
ip link set par_h netns 2815
```

Tabla 3-16: Asignar extremo del part veth al network namespace del contenedor

En la Figura 3-15 vemos que la interfaz ya no está en la lista de las interfaces, en el namespace global y al abrir la terminal de contenedor h1 sí encontramos la interfaz. De modo que gráficamente el escenario se puede ver en la Figura 3-18

```
mayte@virtual-machine:~$ sudo ip link set par_h netns 2815
mayte@virtual-machine:~$ ip link list | grep par_ --color=never
14: par_o@if15: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_f
ast master ovs-system state LOWERLAYERDOWN mode DEFAULT group default qlen
 1000
mayte@virtual-machine:~$ sudo docker exec h1 ip link list
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode D
EFAULT group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
15: par_h@if14: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode
DEFAULT group default qlen 1000
    link/ether f2:fd:28:f8:54:54 brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

Figura 3-15: Asignación extremo del par veth a network namespace del contenedor

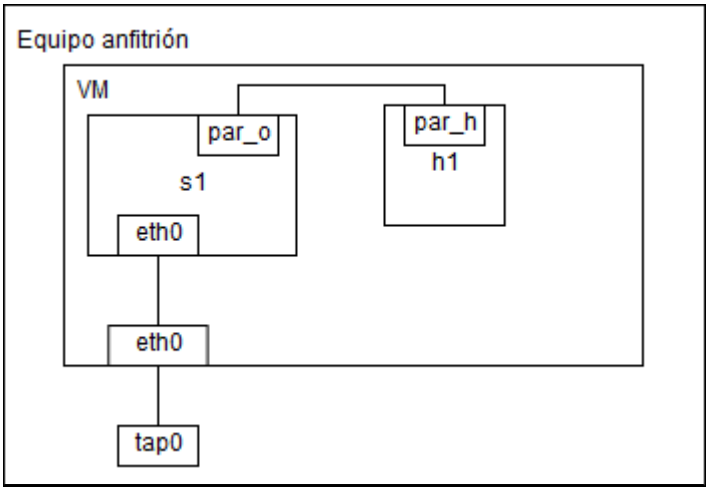

Figura 3-16: Asignación extremo del par veth a network namespace del contenedor

Luego, internamente en el contenedor docker h1 hay que asociar el par_h a la interfaz física eth1 y configurar la IP de la interfaz; esto se puede hacer abriendo la terminal del contenedor con el comando *docker attach h1* como se ha visto en la imagen anterior.

Otra opción, para que sea más sencillo a la hora de automatizar el proceso, es ejecutar los comandos desde el namespace "global" usando el comando *ip netns exec*. Docker por defecto no añade el network namespece del contenedor a los datos de Linux en tiempo de ejecución (linux runtime data /var/run). En caso de que no esté creada se crea la carpeta "netns" que alberga los network namespeces, cuando se crean con el comando ip *netns add <name>*. Luego se enlaza (ln) de manera simbólica (-s) el fichero de network namespace que está en el directorio /proc, con la carpeta "netns" que se ha creado anteriormente generando un fichero de network namespaces, que llamamos con el nombre del PID, en este caso `2815`. El directorio /proc en Linux contiene la jerarquía de ficheros especiales que representan el estado actual del kernel.

Una vez hecho esto ya se puede utilizar el comando *ip netns exec*, con el que se asigna la interfaz par_h a la interfaz física del container eth1, se levanta esta última y se le asigna una IP. A continuación, se muestran los comandos para realizar este proceso y el resultado de la ejecución en la Tabla 3-17 y Figura 3-17. Y quedando un escenario igual al de la versión Mininet Figura 3-9.

```
mkdir -p /var/run/netns

ln -s /proc/2815/ns/net /var/run/netns/2815

ip netns exec 2815 ip link set dev par_h name eth1

ip netns exec 2815 ip link set eth1 up

ip netns exec 2815 ifconfig eth1 192.168.1.221 netmask 255.255.255.0
```

Tabla 3-17: Configuración de la interfaz en el contenedor

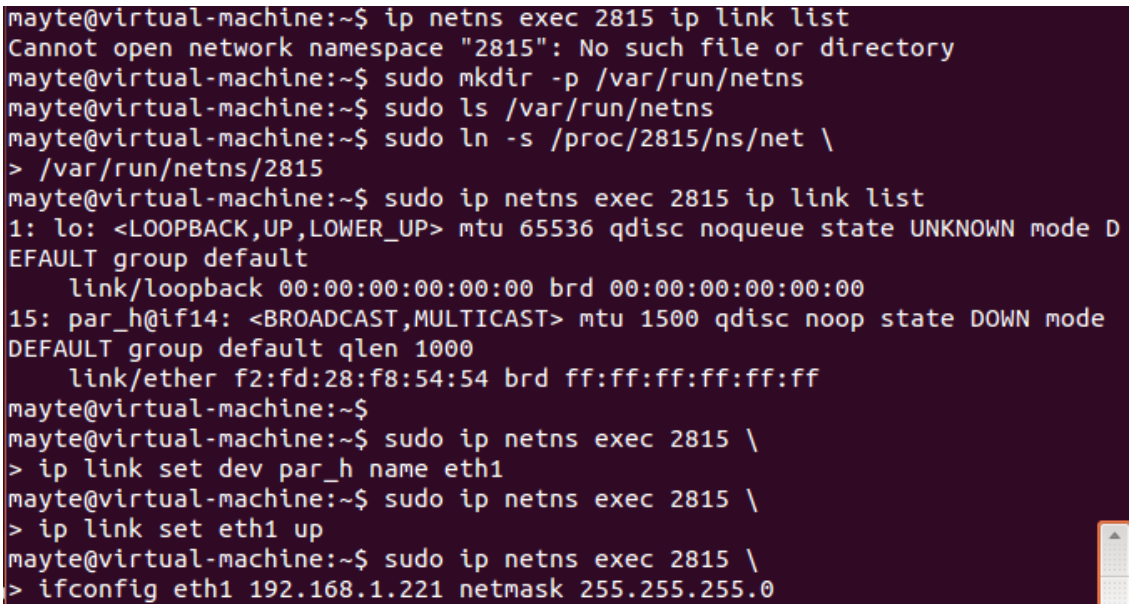

Figura 3-17: Acceso al contenedor desde el anfitrión y configuración interfaz eth1 y par veth

```
mayte@virtual-machine:~$ sudo ip netns exec 2815  ifconfig eth1
eth1      Link encap:Ethernet  direcciónHW f2:fd:28:f8:54:54
          Direc. inet:192.168.1.221  Difus.:192.168.1.255  Másc:255.255.25
5.0
          Dirección inet6: fe80::f0fd:28ff:fef8:5454/64 Alcance:Enlace
          ACTIVO DIFUSIÓN FUNCIONANDO MULTICAST  MTU:1500  Métrica:1
          Paquetes RX:242 errores:0 perdidos:0 overruns:0 frame:0
          Paquetes TX:158 errores:0 perdidos:0 overruns:0 carrier:0
          colisiones:0 long.colaTX:1000
          Bytes RX:43819 (43.8 KB)  TX bytes:22051 (22.0 KB)
```

Figura 3-18: Resultado configuración interfaz en el contenedor

Del mismo modo que en el apartado 3.1 a continuación se configuraría el switch ya sea manualmente o con un controlador para que envíe el tráfico que proceda al host, donde se ejecutará la función de monitorización.

En el Anexo III se muestra la automatización de este proceso con un script *bash*.

3.3.3 Movimiento de NFV

Como último elemento se va a desarrollar un programa que permita mover las funciones de red entre diferentes elementos de la red que funcionen con Linux y puedan ejecutar las funciones de red, es decir, que tengan instalado Open vSwitch y Docker. El objetivo de esto es poder mover la monitorización bajo demanda, al punto de la red que sea necesario, dando así una solución instantánea, más rápida y sencilla, que con sondas físicas.

El programa se va a realizar exclusivamente para Docker, ya que en Mininet, por defecto los hosts o contenedores no tienen aislamiento en el sistema de ficheros, como hemos dicho en la sección 2.4; aunque Mininet proporcione una opción para crear directorios privados. Además, como se ve en esta sección Docker permite fácilmente el control de su demonio de forma remota por lo que se ha considerado que el empleo de Docker es más apropiado para esta tarea.

En primer lugar, hay que ver como instanciar de manera remota los contenedores. En caso de Docker el demonio se puede asociar con una IP y puerto lo que permite controlarlo remotamente. Por defecto el demonio Docker escucha en unix:///var/run/docker.sock para permitir solo conexiones locales de usuarios root. De modo que se configurarán los elementos de red en los que se puedan instanciar las funciones virtuales, para que escuchen a una cierta IP o cualquier equipo poniendo la IP 0.0.0.0, con el siguiente comando:

```
docker daemon -H <IP>:<puerto>
```

Tabla 3-18: Configuración demnio Docker escuchando en <IP> y <Puerto>

Luego desde otro equipo se pueden ejecutar comandos Docker para crear y levantar los contenedores, pudiendo cargar una imagen de contenedor comprimida que esté ubicada en el equipo que ejecuta el comando, como se ve en el siguiente ejemplo.

```
#se carga la imagen docker
docker -H tcp://<IPdst>:<Port> load < dockerimage.tar

#se crea y ejecuta un contenedor
docker -H tcp://<IPdst>:<Port> run --net='none' -i -t -d -h
<usuario> --name <nombreContenedor> dockerimage

#se ejecutan comandos en el contenedor
docker -H tcp://<IPdst>:<Port> exec <nombreContenedor> ifconfig
```

Tabla 3-19: Ejecución remota de Docker

Adicionalmente habrá que conectar el contenedor con el Open vSwitch, que esté creado o crearlo, para que reciba el tráfico que deseamos monitorizar, como ya se ha explicado en las secciones anteriores. Esto se realizará empleando el protocolo SSH a través de la librería de Python paramiko [34] y con este lenguaje se automatizará el proceso del movimiento de las funciones virtuales descrito en esta sección. Las funciones de la librería paramiko que nos interesan en este caso son: crear un cliente SSH y conectarlo al equipo objetivo, como se ve en la Tabla 3-20 y ejecutar comandos en el equipo remoto con privilegios de super-ususario Tabla 3-21.

```
#crear cliente ssh
ssh = paramiko.SSHClient()

#aceptar la host key del dispositivo remoto
ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())

#conectarse con el dispositivo remoto
ssh.connect(ip, username=username,
password=password,look_for_keys=False,allow_agent=False)
```

Tabla 3-20: Conexión SSH paramiko

```
#se envia el comando
stdin, stdout,stderr = ssh.exec_command("sudo -S ovs-vsctl
show")

#se escribe la contraseña
stdin.write(password)
stdin.flush()

#se recupera la salida del comando
data = stdout.read().splitlines()
```

Tabla 3-21: Envío de comandos paramiko

Para que los paquetes lleguen a la sonda de monitorización se configurará el switch OpenFlow al que está conectado, según la función virtual que sea, se configurará el puerto al que está conectado el switch para que haga mirroring, de todo el tráfico Tabla 3-22.

```
#crear port mirror en ovs
ovs-vsctl -- --id=@m create mirror name=mirror0 -- add bridge s1
mirrors @m

#configurar mirror para que envie los paquetes a la interfaz
deseada representa con identificador uuid
ovs-vsctl set mirror mirror0 output_port=<uuid>

#todos los paquetes que reciba el switch hara mirror
ovs-vsctl set mirror mirror0 select_all=1
```

Tabla 3-22: Open vSwitch port mirroring

Para recuperar la información almacenada en los contenedores, hay que encapsularlos, para poder ejecutar la función en una nueva ubicación. Hay que tener en cuenta, como hemos visto en la sección 2.5, que el sistema en capas de Docker, por defecto, no mantiene la capa editable sobre la que se generan los datos de ejecución.

Docker provee dos comandos integrados "docker commit" y "docker save" el primero de los comandos genera una nueva imagen del contenedor, guardando los cambios hechos en las capas editables del contenedor y el segundo comando guarda esa nueva imagen en un fichero comprimido. Finalmente, esa nueva imagen se puede cargar de nuevo en otra instancia de Docker con el comando "docker load". En la Tabla 3-23, se ve un ejemplo de uso de estos comandos.

```
docker commit <contenedor> <nuevaImagen>

docker save <nuevaImagen> > <nuevaImagen.tar>

docker load < <nuevaImagen.tar>
```

Tabla 3-23: Encapsular contenedor Docker

El programa final denominado movenvfv.py realizará las siguientes funciones:

1. Ejecutar una función virtual en un equipo remoto, ejecutando*: python movenfv.py put <IP> <user> <nfv> <image.tar>*. Una vez ejecutado el comando se solicita la contraseña del usuario. En caso de que <nfv> sea la detección de SYN flooding también se solicitará el umbral de detección y la velocidad de refresco, por teclado. El diagrama de flujo de esta función se puede ver en la Figura 3-19.
2. Recuperar el contenedor con su información y extraer los datos recopilados del mismo, ejecutando: *python monvenfv.py get* se mostrará una lista con los contenedores instanciados por el programa previamente. Cada línea que se muestra contendrá la información indicada en la Tabla 3-24. Ejecutando *Python movenfv.py get <nº línea>* se recuperará la función de red y se eliminará en el equipo remoto. Más detalladamente los pasos que realiza son:
    a. Se crea una nueva imagen Docker con "docker commit".

b. Se comprime y recibe en local la imagen con “docker load”.
c. Se para y elimina el contenedor remoto.
d. Se elimina el puerto del OVS al que estuviese conectado.
e. Se crea un contenedor local con la imagen y se extrae el fichero de salida de la sonda.

Campo	**Definición**
nº línea	Número de línea empezando por 0
IP	IP donde se está ejecutando la función de red
OVS	Nombre del bridge al que está conectado
NFV	Función virtual que se está ejecutando
Fecha y hora	Fecha y hora de la ejecución del comando put
Tiempo instanciación	Tiempo que ha tardado en instanciarse en segundos

Tabla 3-24: Campos fichero de log movenfv

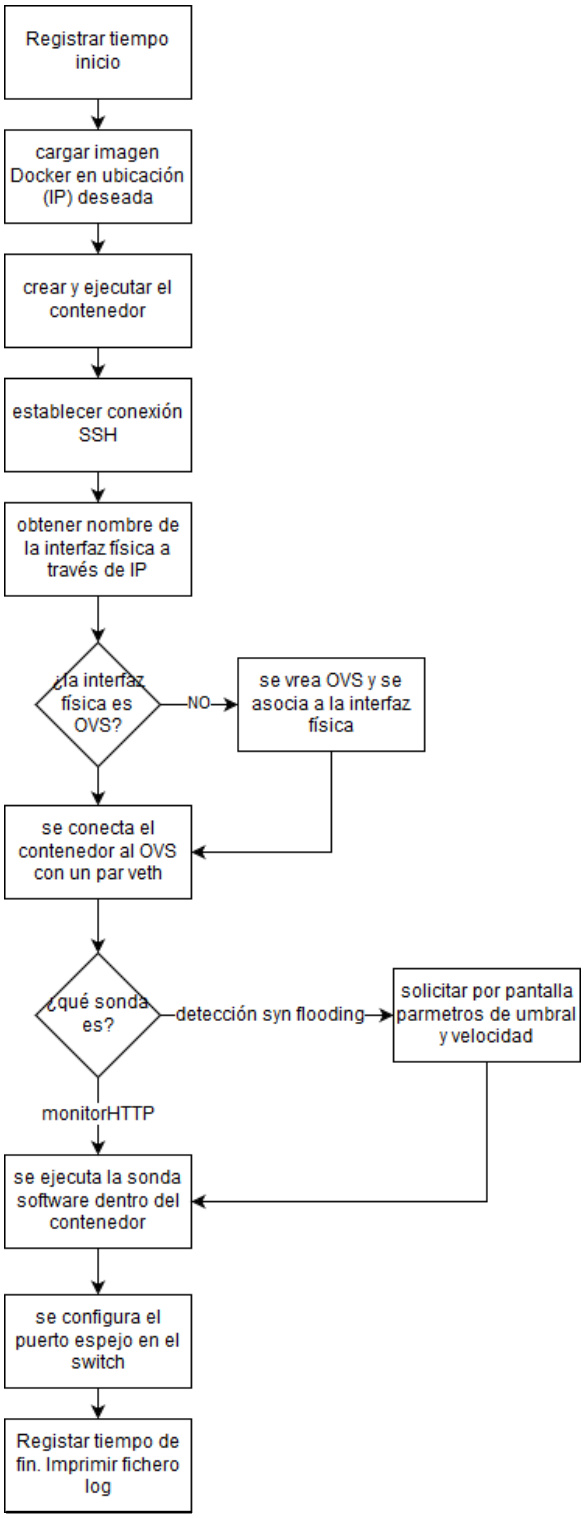

Figura 3-19: Diagrafa de flujo función put de movenfv.py

En el Anexo IV se encuentra el código de esta sección.

4 PRUEBAS Y RESULTADOS

4.1 PRUEBAS DE RENDIMIENTO

Se realizaron pruebas de rendimiento en un escenario igual al esquema inicial de la Figura 3-5. El servidor donde se ejecutaron las pruebas cuenta con 32 GB de RAM, 2 procesadores Intel Xeon E5-2620 con 6 cores cada uno a 2,10 GHz, placa base Supermicro X9DRW y tarjeta de red Intel 82599 10 Gigabit TN. Para generar tráfico se ha usado un equipo con un procesador Intel Xeon E5620 con 4 cores cada uno a 2,40 GHz, 12 GB de RAM y una tarjeta de red Intel 82599 10 Gigabit TN. Para enviar el tráfico se hace uso de la herramienta MoonGen[4] que permite enviar tráfico sintético a velocidades de 10 Gbit/s con paquete de tamaño mínimo.

Primeramente, se ha analizado el troughput alcanzable leyendo paquetes desde dentro de las diferentes soluciones de contenedores (tanto Docker como Mininet) haciendo un *bypass* del tráfico recibido en la tarjeta física de 10 GbE a las interfaces virtuales de dentro de los contenedores. En ambos casos se han obtenido resultados similares como se ve en la Figura 4-1 el ancho de banda que se alcanza no depende de los distintos tipos de contenedores, en ambos casos se alcanza un ancho de banda de en torno a 4.2 Gbit/s con una desviación típica de en torno a 0.3 Gbit/s. Este resultado tiene sentido ya que ambos sistemas de virtualización son muy parecidos y ambos se basan en contenedores. La limitación de ancho de banda que se obtiene teniendo una tarjeta de red de 10 Gigabit está ocasionada por la pila de red de Linux ya que se usa el driver estándar de Intel.

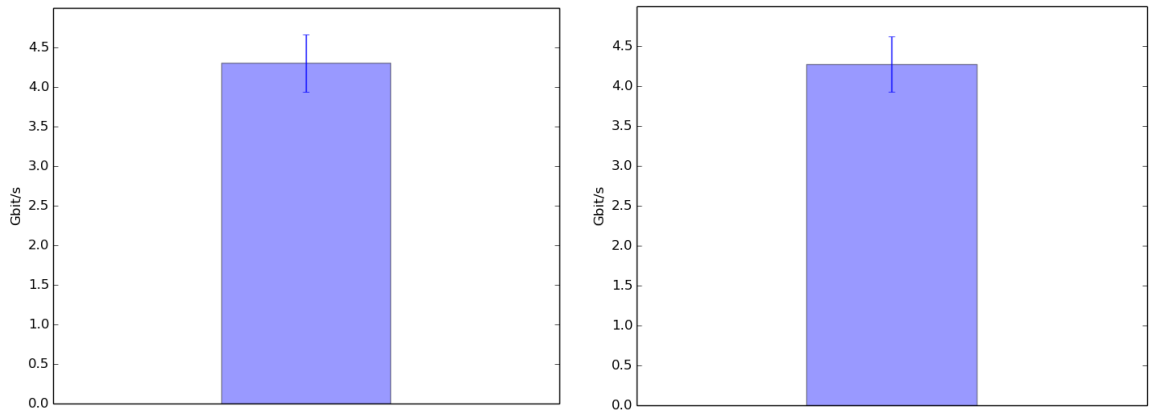

Figura 4-1: Ancho de banda en contenedor Mininet(izq) y Docker (drch) tarjeta de red 10GB

También se evaluó el ancho de banda entre dos contenedores ubicados en un mismo ordenador a través de Open vSwitch. Estas pruebas se han realizado en un ordenador con 8GB de RAM, 4 procesadores Intel Core i5-2520M, con 2 cores cada uno a 2,50GHz, placa base Lenovo 4180WFA y tarjeta de red Intel 82579LM Gigabit Network Connection.

Para realizar la prueba se transmitió una captura de tráfico, con formato pcap, usando tcpreplay, desde uno de los contenedores y se configuró el switch virtual para que retransmitiese todo el tráfico entrante en uno de sus puertos al otro puerto, es decir, al

[4] https://github.com/emmericp/MoonGen

segundo contenedor, donde se midió el ancho de banda con la herramienta ifstat.

En la Figura 4-2 izquierda, se ve el resultado de la prueba realizada con Mininet, al finalizar la transmisión tcpreplay indica que ha transmitido a una tasa media de 3202,69 Mbps y 619452,48 pps, en el otro contenedor se recibe el tráfico a la misma tasa sin pérdidas. Del mismo modo, en la derecha, se ve el resultado de la misma prueba realizada con contenedores Docker. La tasa de transmisión de tcpreplay ha sido de 3388,80 Mbps y 687288,12 pps, en este caso también se ha recibido todo el tráfico, en el contenedor destino, a la misma tasa. Por lo tanto, no debe preocuparnos la perdida de rendimiento internamente dentro de un servidor que albergue las funciones de red.

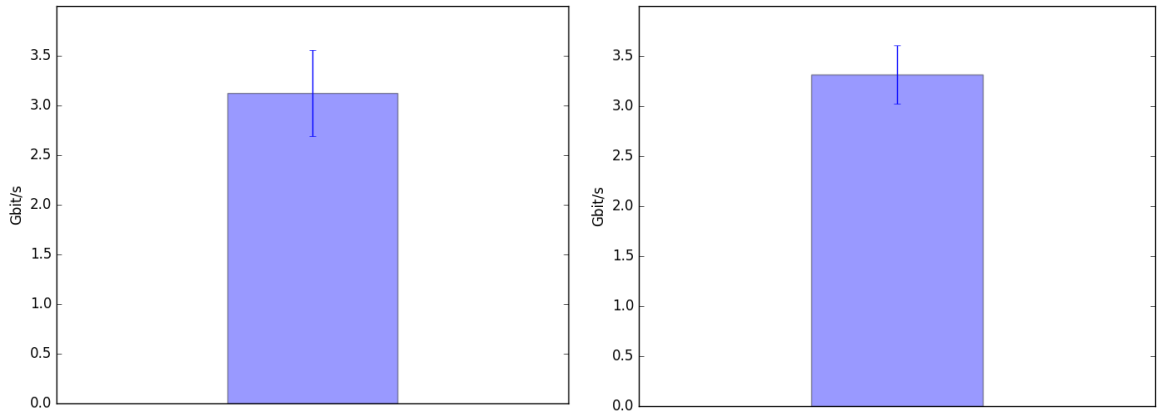

Figura 4-2: Ancho de banda en contenedor Mininet(izq) y Docker (drch) en mismo equipo

4.2 PRUEBAS DE FUNCIONALIDAD

Para validar la funcionalidad de los tres programas desarrollados se ha levantado un escenario como el que se muestra en la Figura 4-3. Consta de tres equipos conectados a un switch OpenFlow, y uno de los tres equipos ejecuta el controlador SDN. Más concretamente, el escenario se monta de forma virtual con VirtualBox y Open vSwitch interconectados mediante interfaces TAP. En la Tabla 4-1, se muestran los datos del escenario, la IP de la interfaz física de cada equipo y datos relevantes sobre la configuración o ejecución de algún elemento.

Nombre	IP	Configuración/Ejecuta
s1	10.0.0.1	ovs-vsctl set-contronller s1 tcp:10.0.0.4:6633
VM1	10.0.0.2	docker daemon -H tcp://0.0.0.0:2375
VM2	10.0.0.3	docker daemon -H tcp:0.0.0.0:2375
VM3	10.0.0.4	./pox.py forwarding.l2_learning openflow.of_01 address=10.0.0.4 port=6633

Tabla 4-1: Datos del escenario

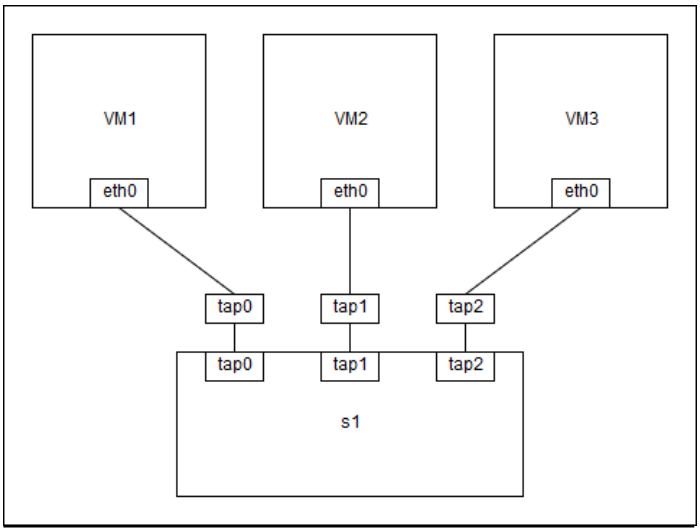

Figura 4-3: Escenario de pruebas

En una primera prueba, partiendo del escenario de la Figura 4-3, se ejecuta desde VM3 el comando: *python movenfv.py put 10.0.0.2 admin monitorHTTP imagen.tar*. Con esto como previamente no hay ningún Open vSwitch conectado a la interfaz física de VM1 se genera uno con el nombre por defecto "brnfv", se crea y conecta el contenedor con la imagen Docker de la función virtual monitorHTTP y comienza a ejecutarse, de modo que el escenario inicial queda con las modificaciones que se ven en la Figura 4-4. Este proceso tarda en realizarse 32,4772401segundos. Desde VM2 se comenzará a enviar tráfico con tcpreplay al que se modifican las direcciones IP y MAC para que cuadren con el escenario mediante tcprewrite.

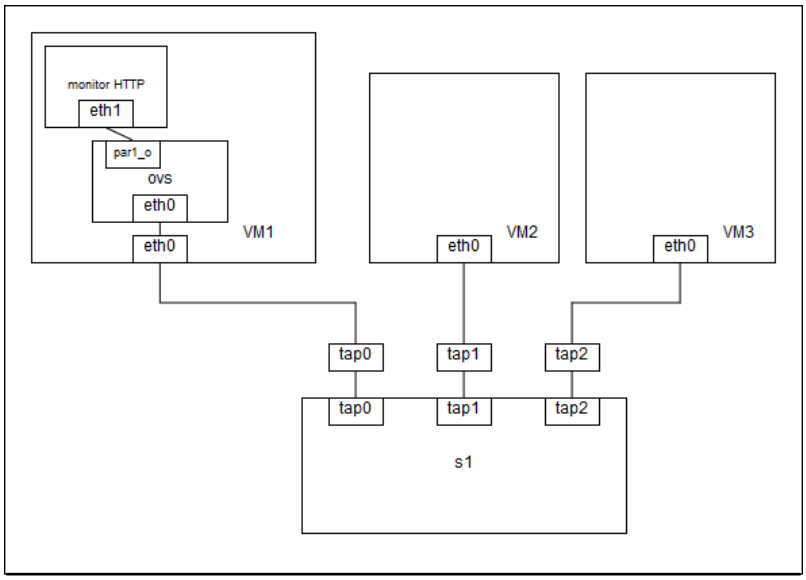

Figura 4-4: Escenario de pruebas con sonda iniciada

Después se, procede a recuperar el contenedor para recuperar la información ejecutando desde VM3 el comando: *python movnfv.py get 0.* Con lo que se para la ejecución del programa monitorHTTP, se almacena el contenedor en una nueva imagen, se recupera el contenedor completo se ejecuta en el equipo local y se extrae el fichero, eliminando el contenedor y la imagen Docker que se creó previamente a excepción del switch que se deja eliminando el puerto de la interfaz del contenedor y la configuración del puerto espejo, volviendo a tener así el escenario de la Figura 4-3. Se comprobó que el fichero de log generado por la sonda concuerda con el tráfico transmitido.

En una segunda prueba, volviendo a partir del escenario de la Figura 4-3, se quiso comprobar cómo se instancia más de una función de red dentro del mismo equipo. De manera análoga a como se realizó en la prueba anterior se ejecutaron las dos funciones de red desarrolladas como se ve en la Figura 4-5, comprobando que simplemente ejecutando la función put del programa movnfv desarrollado se pueden instanciar varias funciones de red ambas funcionando correctamente.

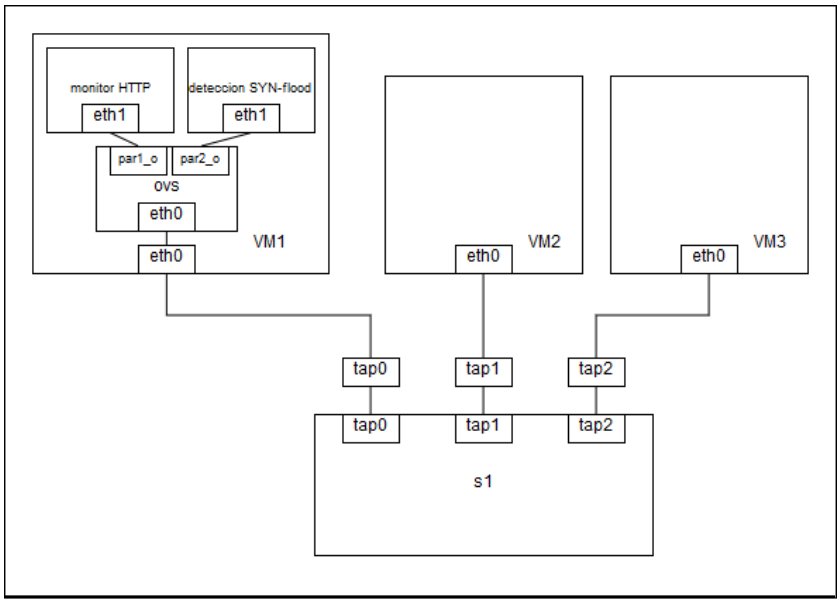

Figura 4-5: Escenario de pruebas con dos sondas iniciadas

5 CONCLUSIONES Y TRABAJO FUTURO

5.1 CONCLUSIONES

En el entorno actual, en el que cada vez hay más dispositivos conectados en red, primando la movilidad y computación en la nube, han surgido dos tendencias complementarias, las redes definidas por software (SDN) y la virtualización de funciones de red (NFV) que permiten abstraerse del hardware, simplificando los elementos de red y centralizando y dinamizando la gestión de las redes.

En este trabajo se ha llevado a cabo un estudio de dos tecnologías de virtualización con el objetivo de integrar en ellas sondas virtuales que faciliten la monitorización ligera en puntos específicos de la red de manera dinámica. En concreto y como muestra del tipo de funciones que se pueden implementar, se han desarrollado dos sondas software para la monitorización de tráfico HTTP y para la detección de ataques por inundación de SYN.

Este trabajo ha hecho uso de tecnologías de virtualización ligera basada en contenedores como son Docker o los que emplea el emulador de red Mininet, causando una carga mínima sobre el sistema que hospeda las funciones virtuales. Esto nos permite mover la monitorización al punto que se desee de la red, evitando así tener que desviar el tráfico, al punto donde se encuentre la función virtual de red, lo que implica aumentar la latencia y modificar los patrones del tráfico que se desean monitorizar. A la hora de instanciar la función de red en el punto que se desee se añade dicha función de red mediante un switch OpenFlow para poder integrarlo dentro de una red SDN y poder gestionar a través del controlador el tráfico que se monitoriza.

Al evaluar el rendimiento de Docker y Mininet conectados a la interfaz física a través de Open vSwitch se ha comprobado que no hay diferencia de rendimiento entre ambas tecnologías, ya que ambas se basan en virtualización de contenedores. Sí se ha observado una limitación del ancho de banda en torno a 4 Gbps que está ocasionada por la pila de red de Linux, ya que se usa el driver estándar de Intel que hace el que el rendimiento de I/O de red se degrade. Por otro lado, el tráfico entre distintos contenedores en un mismo equipo ha transmitido a la tasa de velocidad especificada sin pérdida de paquetes.

Las pruebas de funcionalidad realizadas han demostrado que se pueden ejecutar una o varias funciones de red de manera remota en la ubicación deseada conociendo la IP destino, usuario y contraseña. Posteriormente se puede recuperar el contenedor completo junto con la información contenida en el mismo con el objeto de monitorizar el tráfico en distintas ubicaciones.

Como conclusión personal, con este trabajo he podido comprobar el potencial que ofrecen la virtualización basada en contenedores para la implementación de funciones de red, junto con los switches virtuales Open vSwitch. Además, puesto que estos últimos emplean el protocolo OpenFlow permiten la integración de las funciones virtuales de red en redes definidas por software, lo que aporta una visión centralizada de la red y programabilidad de la misma.

5.2 TRABAJO FUTURO

Como trabajo futuro se plantea la realización de tareas relacionadas con la mejora del rendimiento y la incorporación de nuevas funcionalidades, que inicialmente quedaban fuera del alcance del mismo. Las tareas propuestas son las siguientes:

- **Mejora del rendimiento**:
  - Con el objetivo de alcanzar tasas de 10 Gbps, se propone la utilización de sistemas como DPDK con Open vSwitch para mejorar el rendimiento, permitiendo la copia directa de paquetes de red al espacio de usuario de las aplicaciones, sin interrupciones de hardware.
  - Análisis del tiempo que conlleva la instanciación remota de las funciones virtuales en un entorno con varios nodos, así como el consumo de recursos de CPU y almacenamiento durante la ejecución, con el objetivo de reducir dichos tiempos.
- **Evolución de las funcionalidades**: el desarrollo de un sistema de orquestación de las funciones virtuales de red integrado junto al controlador SDN permitiría encadenar de manera sencillas las funciones virtuales y tener una única herramienta de gestión de la red que tenga control y visión centralizada de la misma. En este trabajo el envío del tráfico hacía las funciones virtuales de red se han realizado configurando puertos espejo, al integrarlo con el controlador la redirección del tráfico se puede realizar instalando reglas en las tablas de flujos de los switches necesarios pudiendo así configurar el tráfico que llega a las funciones de red y simplificando el encadenamiento de las funciones virtuales.

6 REFERENCIAS

[1] J. Bloem, M. Van Doorn, S. Duivestein, D. Excoffier, R. Maas y E. Van Ommeren, «The Fourth Industrial Revolution,» *Sogeti VINT*, 2014.

[2] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar y M. Yu, «SIMPLE-fying middlebox policy enforcement using SDN,» *ACM SIGCOMM computer communication review*, vol. 43, pp. 27-38, 2013.

[3] L. R. Battula, «Network security function virtualization (NSFV) towards cloud computing with NFV over Openflow infrastructure: Challenges and novel approaches,» de *Advances in Computing, Communications and Informatics (ICACCI, 2014 International Conference on*, 2014.

[4] B. A. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka y T. Turletti, «A survey of software-defined networking: Past, present, and future of programmable networks,» *IEEE Communications Surveys \& Tutorials*, vol. 16, pp. 1617-1634, 2014.

[5] O. N. Foundation, «OpenFlow Switch Specification, Version 1.3.0 (Wire Protocol 0x004),» pp. 1-106, 2012.

[6] R. Ozdag, «Intel® Ethernet Switch FM6000 Series-Software Defined Networking,» *See goo. gl/AnvOvX*, p. 5, 2012.

[7] Universidad Autónoma de Madrid, «Escuela Politécnica Superior | Universidad Autónoma de Madrid,» [En línea]. Available: http://www.uam.es/EPS/documento/1242674971538/MEMORIA_INGENIERIA_TELECO_1.pdf?blobheader=application/pdf&blobheadername1=Content-disposition&blobheadername2=pragma&blobheadervalue1=attachment;%20filename=MEMORIA_INGENIERIA_TELECO_1.pdf&blobheadervalue2=pu. [Último acceso: 23 10 2016].

[8] R. Morabito, J. Kjällman y M. Komu, «Hypervisors vs. lightweight virtualization: a performance comparison,» de *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, 2015.

[9] W. Felter, A. Ferreira, R. Rajamony y J. Rubio, «An updated performance comparison of virtual machines and linux containers,» de *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, 2015.

[10] M. Chiosi, D. Clarke, P. Willis, A. Reid, J. Feger, M. Bugenhagen, W. Khan, M. Fargano, C. Cui, H. Deng y othersAAAS, «Network functions virtualisation: An introduction, benefits, enablers, challenges and call for action,» de *SDN and OpenFlow World Congress*, 2012.

[11] B. Han, V. Gopalakrishnan, L. Ji y S. Lee, «Network function virtualization: Challenges and opportunities for innovations,» *IEEE Communications Magazine*,

vol. 53, pp. 90-97, 2015.

[12] E. T. S. I. GSNFV, «Network functions virtualisation (nfv): Architectural framework,» *ETSI GS NFV*, vol. 2, p. V1, 2013.

[13] H. Masutani, Y. Nakajima, T. Kinoshita, T. Hibi, H. Takahashi, K. Obana, K. Shimano y M. Fukui, «Requirements and design of flexible NFV network infrastructure node leveraging SDN/OpenFlow,» de *Optical Network Design and Modeling, 2014 International Conference on*, 2014.

[14] V. Moreno, J. Ramos, P. M. S. del R\́io, J. L. Garc\́ia-Dorado, F. J. Gomez-Arribas y J. Aracil, «Commodity packet capture engines: tutorial, cookbook and applicability,» *IEEE Communications Surveys \& Tutorials*, vol. 17, pp. 1364-1390, 2015.

[15] O. N. Fundation, «Software-defined networking: The new norm for networks,» *ONF White Paper*, 2012.

[16] S. Sezer, S. Scott-Hayward, P. K. Chouhan, B. Fraser, D. Lake, J. Finnegan, N. Viljoen, M. Miller y N. Rao, «Are we ready for SDN? Implementation challenges for software-defined networks,» *IEEE Communications Magazine*, vol. 51, pp. 36-43, 2013.

[17] O. S. Consortium y others, *OpenFlow Switch Specification Version 1.0. 0*, December, 2009.

[18] B. Lantz, B. Heller y N. McKeown, «A network in a laptop: rapid prototyping for software-defined networks,» de *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, 2010.

[19] S.-Y. Wang, C.-L. Chou y C.-M. Yang, «EstiNet openflow network simulator and emulator,» *IEEE Communications Magazine*, vol. 51, pp. 110-117, 2013.

[20] O. vSwitch, «What Is Open vSwitch?,» may 2017. [En línea]. Available: http://docs.openvswitch.org/en/latest/intro/what-is-ovs/.

[21] R. L. S. de Oliveira, C. M. Schweitzer, A. A. Shinoda y L. R. Prete, «Using mininet for emulation and prototyping software-defined networks,» de *Communications and Computing (COLCOM), 2014 IEEE Colombian Conference on*, 2014.

[22] D. Bernstein, «Containers and cloud: From lxc to docker to kubernetes,» *IEEE Cloud Computing*, vol. 1, pp. 81-84, 2014.

[23] L. Containers, «What's LXC?,» may 2017. [En línea]. Available: https://linuxcontainers.org/lxc/introduction/.

[24] D. Inc., «About images, containers, and storage drivers,» may 2017. [En línea]. Available: https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/.

[25] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen y S. Shenker, «Extending Networking into the Virtualization Layer.,» de *Hotnets*, 2009.

[26] P. Emmerich, D. Raumer, F. Wohlfart y G. Carle, «Performance characteristics of virtual switching,» de *Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on*, 2014.

[27] O. vSwitch, «ovs-vsctl - utility for querying and configuring ovs-vswitchd,,» may 2017. [En línea]. Available: http://openvswitch.org/support/dist-docs/ovs-vsctl.8.txt.

[28] O. vSwitch, «Open vSwitch Database Schema,» may 2017. [En línea]. Available: http://openvswitch.org/ovs-vswitchd.conf.db.5.pdf.

[29] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco y F. Huici, «ClickOS and the art of network function virtualization,» de *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, 2014.

[30] R. Cziva, S. Jouet, K. J. S. White y D. P. Pezaros, «Container-based network function virtualization for software-defined networks,» de *2015 IEEE Symposium on Computers and Communication (ISCC)*, 2015.

[31] C. Costache, O. Machidon, A. Mladin, F. Sandu y R. Bocu, «Software-defined networking of linux containers,» de *RoEduNet Conference 13th Edition: Networking in Education and Research Joint Event RENAM 8th Conference, 2014*, 2014.

[32] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach y T. Berners-Lee, «Rfc 2616, hypertext transfer protocol--http/1.1, 1999,» *URL http://www. rfc. net/rfc2616. html*, 2009.

[33] O. vSwitch, «ovs-ofctl - administer OpenFlow switches,,» may 2017. [En línea]. Available: http://openvswitch.org/support/dist-docs/ovs-ofctl.8.txt..

[34] J. Forcier, «Paramiko A Python implementation of SSHv2.,» [En línea]. Available: http://www.paramiko.org/. [Último acceso: 23 4 2017].

[35] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter y G. Shi, «Design and implementation of a consolidated middlebox architecture,» de *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, 2012.

GLOSARIO

NFV: Virtualización de Funciones Re red (Network Function Virtualization), implementación de la función de un middlebox hardware, en software con técnicas de virtualización.

SDN: Red Definida por Software (Software Defined Network), arquitectura de red que se basa en la separación de los planos de control y datos.

IoT: Internet de las cosas (Internet of Things), conexión de elementos contidianos a internet para su monitorización y control, ejemplos del hogar pueden ser control de las luces, consumo de agua, control de electrodomésticos.

SaaS: Software as a Service, Software como Servicio, software bajo demanda no instalado en los equipos del usuario sino en la nube.

PaaS: Platform as a Service, Plataforma como Servicio, proporciona un entorno en la nube sobre el que desarrollar aplicaciones personalizadas.

IaaS: Infraestructure as a Service, Infraestructura como Servicio, proporciona el acceso monitorización y gestión remota de infraestructura de un datacenter, es decir, computo, almacenamiento, y red.

VFNaaS: Virtual Function Network as a Service, en la línea del cloud, la posibilidad del uso de funciones virtuales bajo demanda.

BYOD: Bring Your Own Device, trae tu propio dispositivo, tendencia empresarial por la cual los empleados utilizan sus propios dispositivos en la red corporativa.

IDS: Intrusión Detection Systems, Sistemas de Detección de Intrusión.

NAT: Network Address Translator, Traductores de Direcciones de Red

QoS: Quality of Service, Calidad de Servicio.

CAPEX: CAPital Expenditure, inversiones de capital.

OPEX: OPerational Expenditure, gastos de operación.

TTL: Time To Live, indica por cuantos nodos puede pasar un paquete antes de ser descartado.

VLAN: Virtual Local Area Network, Red de área local virtual.

ANEXOS

Todo el código de estos anexos además se encuentra disponible en el siguiente repositorio web https://bitbucket.org/sdnmon/r_sondasdn.

I MONITORHTTP

```
/*
 * File:   monitorHTTP.c
 * Author: Teresa Pegado
 *
 * Monitoriza tráfico HTTP, impriendo a un fichero
 * -----------------------------------------
 * | timeStamp | IPsrc | IPdst | método | URL |
 * -----------------------------------------
 * metodo y URL son los primeros valores de los datos HTTP separados por \r\n
 * de modo que para las peticiones tenemos método y path (no url completa ...)
 * y para las respuestas HTTP tenemos versión HTTP (HTTP/1.1) y el código de
 * estado
 *
 * Ejecutar:
 * ./monitorHTTP <interfaz>
 */

#include <stdio.h>
#include <../../usr/include/pcap.h>
#include <stdlib.h>
#include <sys/time.h>
#include <string.h>
#include <pthread.h>
#include <unistd.h>
#include <stdint.h>
#include <inttypes.h>
#include <string.h>


#define TAM_TABLA 1000

struct Trama {
    struct timeval timeStamp;
    uint8_t IPsrc[4];
    uint8_t IPdst[4];
    uint8_t PortSrc[2];
    uint8_t PortDst[2];
    char metodo[4];
    char url[10000];
    uint8_t proto[1];

};

size_t tam_tabla = TAM_TABLA;
struct Trama *tabla=NULL;
struct Trama *tabla_n=NULL;


int addTrama(struct Trama trama);
int vacio(struct Trama  trama);
int compararContenido(struct Trama tramaTabla, struct Trama trama);
void limpiar(struct Trama trama);
void manUso(char* argv0);
```

```
/*
 *
 */
int main(int argc, char** argv) {

    struct pcap_pkthdr header;
    const uint8_t *packet;
    uint8_t IPver[1];
    int8_t ihl;
    struct Trama trama;
    uint32_t len =0; //longitud del paquete completo
    uint8_t tcplen =0; //longitud de la cabecera TCP
    int indice = 0;
    int num = 0;
    char *ptr;

    remove("monitorHTTP.txt");

    //reserva de memoria para la tabla
    tabla = (struct Trama *) malloc(tam_tabla*sizeof(struct Trama));

    //abrir fichero pcap
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];

    //parametro de entrada del programa que dispositivo se escucha
    if(argc < 2){
        manUso(argv[0]);
        exit(1);
    }

    //parametro de entrada del programa que dispositivo se escucha
    char *dev = argv[1];

    printf("Device: %s\n", dev);


    //procesar paquetes
    handle = pcap_open_live(dev, BUFSIZ, 0, -1, errbuf);
    //handle = pcap_open_offline("bigFlows.pcap", errbuf); //***
    if (handle == NULL) {
        fprintf(stderr, "pcap_open: %s\n", errbuf);
        exit(0);
    }
    while (1) {
        memset(trama.metodo,0,4);
        memset(trama.url,0,10000);
        if ((packet = pcap_next(handle, &header))) {
            num++;
            trama.timeStamp = header.ts;
            len = header.len;
            //printf("%" PRIu32 "\n",len);
            memcpy(IPver, &packet[14], 1);

            if (((*IPver) >> 4) == 0x4)//IP version 4
            {
                memcpy(trama.IPsrc, &packet[26], 4);
                memcpy(trama.IPdst, &packet[30], 4);
                memcpy(trama.proto, &packet[23], 1);
                ihl = (*IPver) << 4;
                //ihl indica el tamano de la cabecera ip min = 5 max = 15
```

```
                ihl = ihl >> 4;
                memcpy(trama.PortSrc, &packet[14+((ihl*32)/8)],2);
                //cabecera ethernet + ip
                memcpy(trama.PortDst, &packet[16+((ihl*32)/8)],2);
                //cabecera ethernet + ip + hasta portdst

                if (*trama.proto == 6) //si TCP
                {
                    //longitud cabecera tcp
                    indice = 14+((ihl*32)/8)+12;
                    memcpy(&tcplen, &packet[indice], 1);
                    //cabecera ethernet + ip + trozo tcp hasta "Data offset"
                    tcplen = tcplen >> 4;
                    //el data offset son los 4 primeros bits
                    tcplen = (tcplen*32)/8; //paso a bytes
                    //printf("%" PRIu32 "\n",tcplen);
                    //si HTTP
                    indice = 14+((ihl*32)/8)+tcplen;
                    memcpy(trama.metodo, &packet[indice], 4);
                    //cabecera ethernet + ip
                    memcpy(trama.url, &packet[indice], 10000);
                    ptr = strtok( trama.url, "\r\n" );
                    if (ptr != NULL) {
                        if (memcmp(trama.metodo, "GET", 3) == 0 ||
                            memcmp(trama.metodo, "POST", 4) == 0 ||
                            memcmp(trama.metodo, "HTTP", 4) == 0) {
                            //insertar trama en tabla
                            while (addTrama(trama) == 1) {
                            //mientras addTrama devuelve 1 se aumenta
                            // la tabla y se vuelve a añadir
                                tam_tabla = tam_tabla * 2;
                                tabla = (struct Trama *)realloc(tabla,
tam_tabla*sizeof(struct Trama));
                                addTrama(trama);
                            }

                        }
                    }
                }
            }
        }
    }
    pcap_close(handle);
    return;

}


//funcion que añade una trama a la tabla...
//devuelve 0 si se guarda bien, 1 si hay que ampliar la tabla

int addTrama(struct Trama trama) {
    int y;
    int perturb;
    int guardado = 0;
    int yini;
    char *ptr;
    FILE *outfile;


    outfile = fopen("monitorHTTP.txt", "a");
```

```
    //calculo de la posicion a guardar en memoria la quitupla
    y = (*trama.IPsrc + *trama.IPdst + *trama.PortSrc + *trama.PortDst +
*trama.proto) % tam_tabla;
    yini = y;

    //antes de guardar nada comprobar si la posicion esta vacia
    //Si es vacia y es request se guarda
    //Si es vacia y es reponse se imprime
    //Si no es vacia y es request y esta guardado request ?? se da por hecho que
    //si es request del mismo flujo es retransmision por no haber habido reponse
    //se imprime y se guarda nuevo
    //Si no es vacia y es reponse y esta guardado request se imprimen
    //y queda vacio

    while (guardado == 0) {
        if (vacio(tabla[y])) {
            if (memcmp(trama.metodo, "GET", 3) == 0 ||
                memcmp(trama.metodo, "POST", 4) == 0) {//request
                guardado = 1;
                tabla[y] = trama;
            } else {//response
                guardado = 1;
                ptr = strtok(trama.url, "\r\n");
                fprintf(outfile, "----------------------------------------\n");
                fprintf(outfile, "%ld.%06ld\t%hho.%d.%d.%d\t%d.%d.%d.%d\t%s\n",
                   (trama.timeStamp).tv_sec, (trama.timeStamp).tv_usec,
                    trama.IPsrc[0], trama.IPsrc[1], trama.IPsrc[2],
                    trama.IPsrc[3],
                    trama.IPdst[0], trama.IPdst[1], trama.IPdst[2],
                    trama.IPdst[3], ptr);
            }

        } else if (compararContenido(tabla[y], trama)) {
          //comparar contenido en caso se coliciones

            if (memcmp(trama.metodo, "GET", 3) == 0 ||
                memcmp(trama.metodo, "POST", 4) == 0) {//request

                //se imprime request
                ptr = strtok(tabla[y].url, "\r\n");
                fprintf(outfile, "----------------------------------------\n");
                fprintf(outfile, "%ld.%06ld\t%hho.%d.%d.%d\t%d.%d.%d.%d\t%s\n",
                  (tabla[y].timeStamp).tv_sec, (tabla[y].timeStamp).tv_usec,
                   tabla[y].IPsrc[0], tabla[y].IPsrc[1], tabla[y].IPsrc[2],
                   tabla[y].IPsrc[3],
                   tabla[y].IPdst[0], tabla[y].IPdst[1], tabla[y].IPdst[2],
                   tabla[y].IPdst[3], ptr);

                //se guarda nuevo request
                guardado = 1;
                tabla[y] = trama;

            } else {//response
                //se imprime request y response
                ptr = strtok(tabla[y].url, "\r\n");
                fprintf(outfile, "----------------------------------------\n");
                fprintf(outfile, "%ld.%06ld\t%hho.%d.%d.%d\t%d.%d.%d.%d\t%s\n",
                   (tabla[y].timeStamp).tv_sec, (tabla[y].timeStamp).tv_usec,
                   tabla[y].IPsrc[0], tabla[y].IPsrc[1], tabla[y].IPsrc[2],
                    tabla[y].IPsrc[3],
                    tabla[y].IPdst[0], tabla[y].IPdst[1], tabla[y].IPdst[2],
                    tabla[y].IPdst[3], ptr);
```

```
                ptr = strtok(trama.url, "\r\n");
                fprintf(outfile, "%ld.%06ld\t%hho.%d.%d.%d\t%d.%d.%d.%d\t%s\n",
                   (trama.timeStamp).tv_sec, (trama.timeStamp).tv_usec,
                    trama.IPsrc[0], trama.IPsrc[1], trama.IPsrc[2],
                    trama.IPsrc[3],
                    trama.IPdst[0], trama.IPdst[1], trama.IPdst[2],
                    trama.IPdst[3], ptr);

                //vaciar posicoin tabla
                limpiar(tabla[y]);

            }
        } else {
            perturb = y;
            perturb >> 5;
            y = (5 * y) + 1 + perturb;

            y = y % tam_tabla;

            //si el nuevo hash llega a ser igual que el inicial (yini)
            // se amplia la tabla
            if (y == yini) {
                //ampliar tabla
                return 1;
            }
        }
    }

    fclose(outfile);
    return 0;
}


int vacio(struct Trama  trama)
{
    int vacio = 1;
    uint8_t ceros[16]={0};

    if( memcmp( trama.IPdst, ceros, 16 ) ||
            memcmp( trama.IPsrc, ceros, 16 ) ||
            memcmp( trama.PortDst, ceros, 2) ||
            memcmp( trama.PortSrc, ceros, 2) ||
            memcmp( trama.proto, ceros, 1)
            ){
        vacio = 0;
    }

    return vacio;
}

int compararContenido(struct Trama tramaTabla, struct Trama trama)
{
    int igual = 1;
    //si coinciden estos elementos es la misma quintupla y aumentamos el numero
    //de elementos de esa quintupla en 1
    if( memcmp( trama.IPdst, tramaTabla.IPdst, 4 ) ||
            memcmp( trama.IPsrc, tramaTabla.IPsrc, 4 ) ||
            memcmp( trama.PortDst, tramaTabla.PortDst, 2) ||
            memcmp( trama.PortSrc, tramaTabla.PortSrc, 2) ||
            memcmp( trama.proto, tramaTabla.proto, 1) ||
            memcmp( trama.metodo, tramaTabla.metodo, 4)){
```

```
        igual = 0;

    }

    return igual;
}

void limpiar(struct Trama trama) {

    uint8_t ceros[10000] = {0};

    memcpy(trama.IPdst, ceros, 16);
    memcpy(trama.IPsrc, ceros, 16);
    memcpy(trama.PortDst, ceros, 2);
    memcpy(trama.PortSrc, ceros, 2);
    memcpy(trama.proto, ceros, 1);
    memcpy(trama.metodo,ceros,4);
    memcpy(trama.url,ceros,10000);
    trama.timeStamp.tv_sec = 0;
    trama.timeStamp.tv_usec = 0;

}

//funcion para mostrar mensaje inicial de funcionamiento
//en caso de que algun argumento no sea correcto
void manUso(char *argv0){

    printf("Falta algun parametro \nParametros de entrada \n \
 * argv[1] - interfaz de red \n \
 ej: %s eth0\n\n",argv0);

}
```

II MONITOR SYN FLOODING

```
/*
 * File:   monsynflood.c
 * Author: Teresa Pegado
 *
 * Contabilizar el numero de SYN/seg que llega a cada destino desde
multiples origenes
 *
 * Se mide la tasa contando el numero de SYN que llegan a una IP
destino
 * desde distintas IP origenes. Paralelamente se comprueba la cuenta
cada argv[3] segundos
 *
 * Parametros de entrada
 * argv[1] - interfaz de red
 * argv[2] - umbral para considerar tasa SYN/seg como "ataque" e
imprimir resultados.
 * argv[3] - velocidad de refresco de tabla en segundos enteros (int).
Este parametro escala la medida de SYN/seg
 */

#include <stdio.h>
#include <../../usr/include/pcap.h>
#include <stdlib.h>
#include <sys/time.h>
#include <string.h>
#include <pthread.h>
#include <unistd.h>
#include <stdint.h>
//#include <math.h>

typedef struct ListaSrc
{
    uint8_t IPsrc[4];
    int8_t syn;//bit de control del segmento tcp, se utiliza para
               //sincronizar losnumeros de secuencia iniciales de una
               //conexion en el procedimiento de tres fases
    struct ListaSrc *siguiente;
} ListaSrc;

typedef struct TablaDst
{
    uint8_t IPdst[4];
    int8_t syn;//bit de control del segmento tcp, se utiliza para
               //sincronizar losnumeros de secuencia iniciales de una
               //conexion en el procedimiento de tres fases
    struct ListaSrc *lista;
} TablaDst;

typedef struct Quintupla
{
    uint8_t IPdst[4];
    uint8_t PortDst[2];
    uint8_t IPsrc[4];
    uint8_t PortSrc[2];
    uint8_t syn;
} Quintupla;
```

```
#define TAM_TABLA 1000

//variables globales
int tam_tabla = TAM_TABLA;
struct TablaDst *tabla=NULL;
uint8_t end;
FILE *fp;
double synlim;
double velocidad;


pthread_mutex_t lockEnd = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t lockTabla = PTHREAD_MUTEX_INITIALIZER;

int crearIndice(uint8_t *IPdst);
int insertarTabla(struct Quintupla quintupla);
void * comprobarSynseg(void* a);
void manUso(char* argv0);

void * comprobarSynseg(void* a) {
    int i;

    int count = 0;
    struct ListaSrc *nodo = NULL;
    struct ListaSrc *nodoaux = NULL;
    struct timeval tiempo;

    //printf("hilo iniciado\n");
    while (1) {

        pthread_mutex_lock(&lockEnd);
        if (end == 0) {
            fclose(fp);
            break;
        }
        pthread_mutex_unlock(&lockEnd);

        gettimeofday(&tiempo, NULL);

        fp = fopen("monsynflood.txt", "a");

        for (i = 0; i < tam_tabla; i++) {

            pthread_mutex_lock(&lockTabla);

            //Numero de paquetes TCP SYN por segundo que recibe una
            //direccion
            //IP destino desde una direccion IP origen supera umbral
            if (tabla[i].syn > synlim) {
                //Imprimir timestamp
                fprintf(fp,"%ld.%06ld,", tiempo.tv_sec,
tiempo.tv_usec);
                fflush(fp);

                //Imprimir IP destino y SYN total
                fprintf(fp,"%d.%d.%d.%d,%d,",tabla[i].IPdst[0],
                        tabla[i].IPdst[1],tabla[i].IPdst[2],
                        tabla[i].IPdst[3],tabla[i].syn);
                fflush(fp);

                //recorrer lista e imprimir IPs origen y SYN
```

```
                nodo = tabla[i].lista;
                fprintf(fp,"%d.%d.%d.%d,%d,",nodo->IPsrc[0],
                        nodo->IPsrc[1],nodo->IPsrc[2],
                        nodo->IPsrc[3],nodo->syn);
                fflush(fp);
                while(nodo->siguiente != NULL){
                    nodo =  nodo->siguiente;
                    fprintf(fp,"%d.%d.%d.%d,%d,",nodo->IPsrc[0],
                            nodo->IPsrc[1],nodo->IPsrc[2],
                            nodo->IPsrc[3],nodo->syn);
                    fflush(fp);
                    count ++;
                }
                fprintf(fp,"\n");
            }

            //limpiar datos tabla/lista
            if(tabla[i].lista != NULL){
                tabla[i].syn = 0;
                nodo = tabla[i].lista;
                while(nodo->siguiente !=NULL){
                    nodoaux = nodo;
                    nodo = nodoaux->siguiente;
                    free(nodoaux);
                }
                tabla[i].lista = NULL;
            }

            pthread_mutex_unlock(&lockTabla);
        }
        sleep(velocidad); //cambia a 10 seg
    }
}

int main(int argc, char** argv) {

    struct pcap_pkthdr header;
    const uint8_t *packet;
    uint8_t IPver[1];

    //datos guardar
    Quintupla quintupla;
    uint8_t syn_aux;
    uint8_t proto[1];
    uint8_t ihl;
    uint8_t flags[1];

    pthread_t id;
    int ret;

    //borrar fichero de salida de una ejecucion anterior
    remove("ataqueDOS.txt");


    //abrir fichero pcap
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];

    //parametro de entrada del programa que dispositivo se escucha
    if(argc < 4){
        manUso(argv[0]);
```

```
        exit(1);
    }

    char *dev = argv[1];
    synlim = atof(argv[2]);
    velocidad = atof(argv[3]);

    printf("Escuchando en %s\n", dev);
    printf("Umbral de SYN/seg %f\n",synlim);
    printf("Velocidad hilo %f\n", velocidad);

    synlim = synlim * velocidad;

    //reserva de memoria para la tabla
    tabla = (struct TablaDst *) malloc(tam_tabla*sizeof(struct
TablaDst));
    if(tabla == NULL){
        printf("Error al reservar memoria para la tabla");
        exit(0);
    }
    tabla->lista =  NULL;

    end = 1;
    //creo hilo para contador
    ret=pthread_create(&id, NULL, comprobarSynseg,NULL);
    if(ret!=0)
    {
        printf("Error al crear hilo");
        exit(0);
    }

    handle = pcap_open_live(dev, BUFSIZ, 0, -1, errbuf);
    //handle = pcap_open_offline("bigFlows.pcap", errbuf);
    if (handle == NULL) {
        fprintf(stderr, "pcap_open: %s\n", errbuf);
        exit(0);
    }

    while (1) {
        if ((packet = pcap_next(handle, &header))) {


            //IP version 4
            memcpy(IPver, &packet[14], 1);
            if (((*IPver) >> 4) == 0x4) {

                memcpy(quintupla.IPsrc, &packet[26], 4);
                memcpy(quintupla.IPdst, &packet[30], 4);

                memcpy(proto, &packet[23], 1);
                ihl = (*IPver) << 4; //ihl indica el tamano de la
                                     //cabecera ip min = 5 max = 15
                ihl = ihl >> 4;
                memcpy(quintupla.PortSrc, &packet[14 + ((ihl * 32) /
8)], 2);
                //cabecera ethenet + ip
                memcpy(quintupla.PortDst, &packet[16 + ((ihl * 32) /
8)], 2);
                //cabecera ethernet + ip + hasta portdst

                if (proto[0] == 6) {
```

```
                    memcpy(flags, &packet[27 + ((ihl * 32) / 8)], 1);
                    //extraer bit syn de flag
                    syn_aux = (int) *flags & 2;//selecciono con un
                                               //AND el bit de SYN (A)
                    quintupla.syn = (syn_aux >> 1)&0x01;
                      //lo desplazo uno a la derecha
                } else {
                    quintupla.syn = 0;
                }
                if (quintupla.syn == 1) {

                    pthread_mutex_lock(&lockTabla);
                    while (insertarTabla(quintupla) == 1) {
                    //mientras addTrama devuelve 1 se aumenta la tabla
                    //y se vuelve a anadir
                        tam_tabla = tam_tabla * 2;
                        tabla = (struct TablaDst *) realloc(tabla,
tam_tabla*sizeof(struct TablaDst));
                        insertarTabla(quintupla);
                    }
                    pthread_mutex_unlock(&lockTabla);
                }
            }

        }
    }

    pcap_close(handle);
    free(tabla);
    end = 0;
    return;
}

//funcion que inserta/actualiza en la tabla los datos de un nuevo
//paquetes
//devuelve 0 si se guarda bien, 1 si hay que ampliar la tabla
int insertarTabla(struct Quintupla quintupla) {

    int ind;
    int perturb;
    int indini;
    int count = 0;
    int insertado = 0;
    struct ListaSrc *nodo = NULL;
    struct ListaSrc *nodoant = NULL;
    uint8_t ipvacia[4] = {0};

    //Crear nuevo nodo para la lista
    nodo = (struct ListaSrc *)malloc(sizeof(struct ListaSrc));

    ind = (quintupla.IPdst[0] + quintupla.IPdst[1] +
           quintupla.IPdst[2] + quintupla.IPdst[3])%tam_tabla;
    indini = ind;
    //insertar datos sobre IP dst en tabla
    //mientras no se inserten los datos
        while(insertado == 0){
             if(memcmp(tabla[ind].IPdst, ipvacia, 4) == 0 ){
                //entrada vacia, utilizar
                memcpy(tabla[ind].IPdst, quintupla.IPdst, 4);
                tabla[ind].syn = tabla[ind].syn + quintupla.syn;
                insertado = 1;
```

```
            }else if(memcmp(tabla[ind].IPdst, quintupla.IPdst, 4) == 0
){
                //entrada ocupada IPs coinciden, actualizar datos
                tabla[ind].syn = tabla[ind].syn + quintupla.syn;
                insertado = 1;
            }else{
                // colision recalcular hash
                perturb = ind;
                ind = (5*ind)+1+perturb;
                perturb >> 5;
                ind = ind % tam_tabla;
                //ind++;
                if(ind == indini){
                    return 1;
                }
            }
        }

        //insertar datos sobre IP src en lista enlazada
        insertado = 0;
        while (insertado == 0) {
            if (tabla[ind].lista == NULL) {
                //lista esta vacia se rellena el primer nodo
                nodo->siguiente = tabla[ind].lista;
                tabla[ind].lista = nodo;

                memcpy(nodo->IPsrc, quintupla.IPsrc,4);
                nodo->syn = quintupla.syn;
                insertado = 1;
            }else{
                //lista no esta vacia
                //primera iteracion, avanzar a primer nodo
                if(count == 0)
                    nodoant = tabla[ind].lista;

                if(memcmp(nodoant->IPsrc,quintupla.IPsrc,4) == 0){
                    //IPs de los nuevos datos coinciden, actualizar
                    //conteo syn
                    nodoant->syn = nodoant->syn + 1;
                    insertado = 1;
                }else if(nodoant->siguiente ==  NULL){
                    //No hay siguiente nodo, se anade uno nuevo
                    memcpy(nodo->IPsrc, quintupla.IPsrc,4);
                    nodo->syn = nodo->syn + quintupla.syn;
                    nodo->siguiente = nodoant->siguiente;
                    nodoant->siguiente = nodo;
                    insertado  =1;
                }else{
                    //Se avanza al siguiente nodo
                    nodoant = nodoant->siguiente;
                    count ++;
                }

            }


        }


    return 0;
}
```

```
//funcion para mostrar mensaje inicial de funcionamiento en caso de
que algun argumento no sea correcto
void manUso(char *argv0){

    printf("Falta algun parametro \nParametros de entrada \n \
 * argv[1] - interfaz de red \n \
 * argv[2] - umbral para considerar tasa SYN/seg como ataque e
imprimir resultados. \n \
 * argv[3] - velocidad de refresco de tabla en segundos enteros (int).
Este parametro escala la medida de SYN/seg \n \
 ej: %s eth0 4 1\n\n",argv0);

}
```

```
#!/bin/bash

#

FOUND=`grep "s1" /proc/net/dev`
if  [ -n "$FOUND" ] ; then
    #echo "s1 ya existe, eliminalo y ejecuta de nuevo"
    ovs-vsctl del-br s1
    ip link del "s1-h1-h"
fi


ovs-vsctl add-br s1
ovs-vsctl add-port s1 eth0
ifconfig eth0 0
ifconfig s1 192.168.1.220 netmask 255.255.255.0 #ip del equipo
route add default gw 192.168.1.1 s1


FOUND1=`docker ps | grep h1`
FOUND2=`docker ps -a | grep h1`

if [ -n "$FOUND1" ] ; then
   docker stop h1
elif [ -n "$FOUND2" ] ; then
   docker build -t bwmes .
   docker run --net='none' -i -t --name h1 bwmes
fi

docker start h1

ip link add "s1-h1-h" type veth peer name "s1-h1-c"

ovs-vsctl add-port s1 "s1-h1-h" -- set interface "s1-h1-h"
external_ids:container_id="h1" external_ids:container_iface=eth1

ip link set "s1-h1-h"

if PID=`docker inspect -f '{{.State.Pid}}' "h1"`; then :; else
echo >&2 "$UTIL: Failed to get the PID of the container"
exit 1
fi

echo "$PID"
ip link set "s1-h1-c" netns "$PID"
mkdir -p /var/run/netns



ln -s /proc/"$PID"/ns/net /var/run/netns/"$PID"
ip netns exec "$PID" ip link set dev "s1-h1-c" name eth1
ip netns exec "$PID" ip link set eth1 up
ip netns exec "$PID" ifconfig eth1 192.168.1.222 netmask 255.255.255.0

ovs-ofctl del-flows s1
ovs-ofctl add-flow s1 in_port=1,actions=output:2
ovs-ofctl add-flow s1 in_port=2,actions=output:1
```

IV MOVIMIENTO NFV

```
#File: movenfv.py

#Author: M Teresa Pegado
#
#Script para enviar (put) y recuperar (get) nfv
#          movenfv.py put <IP> <username> <nfv> <dockerImage.tar>
#             IP: IP equipo remoto
#             username: nombre de usuario
#             nfv: funcion virtual a ejecutar (monitorHTTP o
#                  synfloodDeteccion)
#             dockerImage.tar: fichero comprimido .tar con la image
#                              Docker
#          movenfv.py get # muestra la lista de funciones ejecutadas
#                           previamente
#          movenfv.py get <n linea> #recupera la funcion de la linea
#                                    que se indique

import paramiko
import subprocess
import time
import datetime
import os
import sys
import getpass


def put_nfv(ip, username,password, nfv, dockerImageTar):

   tic = time.time()
   ##cargar imagen docker
   comando = "docker -H tcp://"+ip+":2375 load < "+dockerImageTar
   print comando
   os.system(comando)

   ##crear y ejecutar contenedor
   comando = "docker -H tcp://"+ip+":2375 run --net='none' -i -t -d -h
"+nfv+" --name "+nfv+" "+dockerImageTar[:-4]
   print comando
   os.system(comando)

   #se establece la conexion ssh
   ssh = paramiko.SSHClient()
   ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
   ssh.connect(ip, username=username,
password=password,look_for_keys=False,allow_agent=False)

   #obtengo nombre de la interfaz fisica a partir de la IP
   stdin, stdout,stderr = ssh.exec_command("ifconfig")
   data = stdout.read().splitlines()
   for line in data:
      found = line.find("Ethernet")
      found1 = line.find("inet:")
      if(found !=-1):
         lineaux = line.split(" ")
         interfaceaux = lineaux[0]

      elif(found1 != -1):
         if(line.find(ip) != -1):
            interface = interfaceaux
```

```
   #se comprueba si la interfaz fisica es el switch
   stdin, stdout,stderr = ssh.exec_command("sudo -S ovs-vsctl show")
   stdin.write(password+'\n')
   stdin.flush()
   data = stdout.read()
   print data
   if(data.find(interface) != -1):
      print "la interfaz fisica es un ovs  "+interface
      nombreBridge = interface
      print interface
   else:
      print "interfaza fisica no es ovs"
      # se crea un ovs y se conecta a la interfaz fisica
      nombreBridge = "brnfv"
      stdin, stdout,stderr = ssh.exec_command('sudo -S ovs-vsctl add-
br "'+nombreBridge+'"')
      stdin.write(password+'\n')
      stdin.flush()

      stdin, stdout,stderr = ssh.exec_command("sudo -S ovs-vsctl add-
port "+nombreBridge+" "+interface+" && sudo -S ifconfig "+interface+"
0 && sudo -S ifconfig "+nombreBridge+" "+ip+" netmask 255.255.255.0")
      stdin.write(password+'\n')
      stdin.flush()

      stdin, stdout,stderr = ssh.exec_command("sudo -S ovs-vsctl set-
controller "+nombreBridge+" "+"tcp:10.0.0.4:6633")
      stdin.write(password+'\n')
      stdin.flush()

   #se crea par veth
   comando = 'sudo -S ip link add "par_'+nfv[-4:]+'h" type veth peer
name "par_'+nfv[-4:]+'o"'
   print comando
   stdin, stdout,stderr = ssh.exec_command(comando)
   stdin.write(password+'\n')
   stdin.flush()


   comando = 'sudo -S ovs-vsctl add-port '+nombreBridge+' "par_'+nfv[-
4:]+'o"'
   print comando
   stdin, stdout,stderr = ssh.exec_command(comando)
   stdin.write(password+'\n')
   stdin.flush()

   comando = 'sudo -S ip link set "par_'+nfv[-4:]+'o" up'
   print comando
   stdin, stdout,stderr = ssh.exec_command(comando)
   stdin.write(password+'\n')
   stdin.flush()

   comando = "docker -H tcp://"+ip+":2375 inspect -f '{{.State.Pid}}'
"+nfv
   print comando
   p = subprocess.Popen(comando, shell=True, stdout=subprocess.PIPE,
stderr=subprocess.PIPE)
   out, err = p.communicate()
   print out
   data = out.splitlines()
```

```
    for line in data:
       print line
    print data
    pid = data[0]

    comando = 'sudo -S ip link set "par_'+nfv[-4:]+'h" netns '+pid
    print comando
    stdin, stdout,stderr = ssh.exec_command(comando)
    stdin.write(password+'\n')
    stdin.flush()
    data = stdout.read()
    print data
    err = stderr.read()
    print err

    comando = "sudo -S mkdir -p /var/run/netns"
    print comando
    stdin, stdout,stderr = ssh.exec_command(comando)
    stdin.write(password+'\n')
    stdin.flush()
    data = stdout.read()
    print data
    err = stderr.read()
    print err

    comando = "sudo -S ln -s /proc/"+pid+"/ns/net /var/run/netns/"+pid
    print comando
    stdin, stdout,stderr = ssh.exec_command(comando)
    stdin.write(password+'\n')
    stdin.flush()
    data = stdout.read()
    print data
    err = stderr.read()
    print err

    comando = 'sudo -S ip netns exec '+pid+' ip link set dev
"par_'+nfv[-4:]+'h" name eth1'
    print comando
    stdin, stdout,stderr = ssh.exec_command(comando)
    stdin.write(password+'\n')
    stdin.flush()
    data = stdout.read()
    print data
    err = stderr.read()
    print err

    comando = "sudo -S ip netns exec "+pid+" ip link set eth1 up"
    print comando
    stdin, stdout,stderr = ssh.exec_command(comando)
    stdin.write(password+'\n')
    stdin.flush()
    data = stdout.read()
    print data
    err = stderr.read()
    print err

    #arrancar nfv
    if nfv=="monitorHTTP":
       comando = "docker -H tcp://"+ip+":2375 exec -d "+nfv+"
./tmp/"+nfv+" eth1"
       print comando
```

```
    elif nfv=="monsynflood":
        umbral = input(" Introduce umbral de SYN/seg: ")
        velocidad = input("Introduce la velocidad de comprobacion en
segundos: ")
        comando = "docker -H tcp://"+ip+":2375 exec -d "+nfv+"
./tmp/"+nfv+" eth1 "+str(umbral)+" "+str(velocidad)
        print comando
    else:
        print "*****************************************************"
    os.system(comando)

    #comprobar que se esta ejecutando
    comando = "docker -H tcp://"+ip+":2375 exec "+nfv+" ps"
    print comando
    os.system(comando)

    #crear port mirror en ovs
    comando = "sudo -S ovs-vsctl -- --id=@m create mirror name="+nfv[-
4:]+" -- add bridge "+nombreBridge+" mirrors @m"
    print comando
    stdin, stdout,stderr = ssh.exec_command(comando)
    stdin.write(password+'\n')
    stdin.flush()

    #encontrar UUID de la interfaz del container
    comando = 'sudo -S ovs-vsctl list port "par_'+nfv[-4:]+'o"'
    print comando
    stdin, stdout,stderr = ssh.exec_command(comando)
    stdin.write(password+'\n')
    stdin.flush()
    data = stdout.read().splitlines()
    for line in data:
        found = line.find("_uuid")
        if(found!=-1):
            lineaux = line.split(":")
            uuidaux1 = lineaux[-1]
            uuidaux2 = uuidaux1.split(" ")
            print uuidaux2
            uuid = uuidaux2[-1]
            print "uuid: "+uuid

    #configurar mirror para que envie los paquetes a la interfaz
    comando = "sudo -S ovs-vsctl set mirror "+nfv[-4:]+"
output_port="+uuid
    print comando
    stdin, stdout,stderr = ssh.exec_command(comando)
    stdin.write(password+'\n')
    stdin.flush()

    #todos los paquetes que reciba el switch hara mirror
    comando = "sudo -S ovs-vsctl set mirror "+nfv[-4:]+" select_all=1"
    print comando
    stdin, stdout,stderr = ssh.exec_command(comando)
    stdin.write(password+'\n')
    stdin.flush()

    #segundos que ha tardado en instanciarse
    toc = time.time() - tic
    print "nfv ejecutandose instanciacion a tardado "+str(toc)+"
segundos"
    now = datetime.datetime.now()
```

```
    f = open('logmovenfv.txt', 'a')

f.write(ip+";"+nombreBridge+";"+nfv+";"+str(now)+";"+str(toc)+";"+user
name+";"+password+"\n")
    f.close()

    ssh.close()

def get_nfv(ip, nombreBridge, nfv, username, password):

    #crear nueva imagen docker a partir existente
    comando = "docker -H tcp://"+ip+":2375 commit "+nfv+" rec"
    print comando
    os.system(comando)

    #comprimir imagen y recibir
    comando = "docker -H tcp://"+ip+":2375 save rec > rec.tar"
    print comando
    os.system(comando)

    #para contenedor en destino
    comando = "docker -H tcp://"+ip+":2375 stop "+nfv
    print comando
    os.system(comando)

    #eliminar contenedor en destino
    comando = "docker -H tcp://"+ip+":2375 rm "+nfv
    print comando
    os.system(comando)

    #se establece la conexion ssh
    ssh = paramiko.SSHClient()
    ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    ssh.connect(ip, username=username,
password=password,look_for_keys=False,allow_agent=False)

    #elimino el puerto del ovs
    comando = 'sudo -S ovs-vsctl del-port '+nombreBridge+' "par_'+nfv[-
4:]+'o"'
    print comando
    stdin, stdout,stderr = ssh.exec_command(comando)
    stdin.write(password+'\n')
    stdin.flush()

    #eliminar mirror
    comando = 'sudo -S ovs-vsctl -- --id=@m get mirror '+nfv[-4:]+' --
remove bridge '+nombreBridge+' mirrors @m'

    #cargar imagen en local
    comando = "docker load < rec.tar"
    print comando
    os.system(comando)

    #crear y ejecutar contenedor
    comando = "docker run --net='none' -i -t -d -h "+nfv+" --name
"+nfv+" rec"
    print comando
    os.system(comando)

    #extraer fichero
    comando = "docker cp "+nfv+":/"+nfv+".txt "+nfv+".txt"
```

```
    print comando
    os.system(comando)

    #para contenedor
    comando = "docker stop "+nfv
    print comando
    os.system(comando)

    #eliminar contenedor
    comando = "docker rm "+nfv
    print comando
    os.system(comando)

    #eliminar imagen
    comando = "docker rmi rec"
    print comando
    os.system(comando)


    ssh.close()



def readLog():
    tabla = []
    f = open('logmovenfv.txt', 'r')
    i = 0
    for line in f:
        lineaux = line.split(";")
        tabla.append(lineaux)
        print(str(i)+" "+str(lineaux[0:5]))
        i=i+1
    f.close()

def callget(posicion):
    tabla = []
    f = open('logmovenfv.txt', 'r')
    i = 0
    for line in f:
        lineaux = line.split(";")
        if lineaux!="\n":
            #print lineaux
            tabla.append(lineaux)
    f.close()
    ip = tabla[int(posicion)][0]
    nombreBridge = tabla[int(posicion)][1]
    nfv = tabla[int(posicion)][2]
    username = tabla[int(posicion)][5]
    password = tabla[int(posicion)][6]
    password = password.split("\n")
    password = password[0]

    del tabla[int(posicion)]
    f = open('logmovenfv.txt', 'w')
    for i in tabla:
        f.write(str(i[0])+";"+str(i[1])+";"+str(i[2])+";"+\
                str(i[3])+";"+str(i[4])+";"+str(i[5])+";"+str(i[6]))
    f.close()

    get_nfv(ip,nombreBridge,nfv,username,password)
```

```
def showinfo():
   print "Script para enviar (put) y recuperar (get) nfv\n\n \
          movenfv.py put <IP> <username> <nfv> <dockerImage.tar>\n \
             IP: IP equipo remoto\n \
             username: nombre de usuario\n \
             nfv: funcion virtual a ejecutar (monitorHTTP o
                  synfloodDeteccion)\n \
             dockerImage.tar: fichero comprimido .tar con la image
                              Docker\n\n \
          movenfv.py get # muestra la lista de funciones ejecutadas
                           previamente\n\n \
          movenfv.py get <n linea> #recupera la funcion de la linea
                                    que se indique\n"


##MAIN

if len(sys.argv) < 2:
   print "No se ha indicado ninguna funcion"
   showinfo()
else:
   if sys.argv[1] == 'put':
      if len(sys.argv) != 6:
        print "Numero de argumentos no valido"
        showinfo()
      else:
        print "Introduce la contrasena del equipo destino:\n"
        pw = getpass.getpass()
        put_nfv(sys.argv[2], sys.argv[3], pw, sys.argv[4],
sys.argv[5])
   elif sys.argv[1] == 'get':
      if len(sys.argv) != 3:
         print "Falta un argumento indica la linea a eliminar\n \
                la numeracion empieza por 0 \n \
                movenfv.py get <n linea>\n"
         readLog()
      else:
         print sys.argv[2]
         callget(sys.argv[2])
   else:
      print "Esa funcion no existe"
      showinfo()
```